RL-TR-96-279
In-House Report
February 1997

DYNAMICALLY RECONFIGURABLE FPGA-BASED MULTIPROCESSING AND FAULT TOLERANCE

Kevin A. Kwiat

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-96-279 has been reviewed and is approved for publication.

APPROVED: Eugene C. Blackburn

EUGENE C. BLACKBURN
Chief, Electronics Reliability Division
Electromagnetics & Reliability Directorate

FOR THE COMMANDER: John J. Bart

JOHN J. BART
Chief Scientist, Reliability Sciences
Electromagnetics & Reliability Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 1997	In-House Aug 93 - Jul 96

4. TITLE AND SUBTITLE

DYNAMICALLY RECONFIGURABLE FPGA-BASED MULTIPROCESSING AND FAULT TOLERANCE

5. FUNDING NUMBERS

PE - 62702F
PR - 2338
TA - 01
WU - 7B

6. AUTHOR(S)

Kevin A. Kwiat

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Rome Laboratory/ERDA
525 Brooks Road
Rome, NY 13441-4505

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Rome Laboratory/ERDA
525 Brooks Road
Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-96-279

11. SUPPLEMENTARY NOTES

Also published as a doctoral dissertation in computer engineering at Syracuse University.
Project Engineer: Kevin Kwiat/ERDA, 315-330-4635

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; Distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Processor replication is a straightforward yet general method for contending with a range of failure modes in computing systems. Hardware-intensive solutions lower utilization by dedicating resources solely to fault tolerance. Alternately, software-intensive solutions, although flexible, require coping with slower speeds.
Our objective in this research is to eliminate these problems and limitations by adding a multiprocessing capability so that the redundant hardware is used selectively. The architecture we propose is flexible in that processors do not have to be tightly synchronized, and it permits unconstrained combinations of fault-tolerant and multiprocessing applications to execute concurrently for efficient processor utilization. We capitalize on dynamically reconfigurable Field-Programmable Gate Array (FPGA) technology, which permits changing portions of its logic without disturbing the rest of the array. We demonstrate our tools and techniques for reconfiguring the FPGA, even while it operates, to create a virtual FPGA that is much larger than the physical FPGA. We have analyzed the proposed architecture and evaluated it with respect to performance, cost, scalability, reliability, and configurability. Based on all of these criteria, we have shown that our approach is superior to comparable designs.

14. SUBJECT TERMS

Fault Tolerance, Fault-Tolerant Computing, Multiprocessing, Field Programmable Gate Array, FPGA, Dynamic Reconfiguration

15. NUMBER OF PAGES

218

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	U/L

Contents

List of Figures

Acknowledgments

I thank Dr. Salim Hariri, my advisor, for his guidance and patience throughout this work. Without his criticism and encouragement, this work could not have been completed. It was a pleasure working with him.

Other members of the Syracuse University faculty also assisted me. Dr. Ed Stabler served as reader of this dissertation. Dr. Shui-Kai Chin, Dr. Shihab Ghaya, Dr. Carlos Hartmann, and Dr. Pramod Varshney participated as dissertation committee members.

I am grateful for the support given to me by the United States Air Force and Rome Laboratory for this research. I thank my mentor, Dr. Warren Debany, for his constancy and reinforcement.

My special thanks go to my wife Trish for her patience during these past years. Without her support, I could not have possibly completed this undertaking. I hope my three children, Hayley, Adam, and Luke will understand their father's distraction from giving them more attention during this time. My mother-in-law and father-in-law both deserve much credit for their assistance. My thanks also go to my brothers – especially Mark, who guided and inspired me to study computer technology. Finally, I want to dedicate this dissertation and degree to my mother and father, who always gave me encouragement and endless sacrifice.

Chapter 1

Introduction

The origins of fault-tolerant computing, for electronic computers, is credited to von Neumann [1]. In 1837, Charles Babbage wrote [2] that a complicated formula could be algebraically arranged in several ways such that, if the same values are assigned to the variables and the results agree, then the accuracy of the computation is secure. Babbage, of course, was referring to the work of clerical staff — the "computers" of his time. Even with its long history, fault-tolerant computing is undergoing continuing development today as advances in digital computer technology result in applications that call for high reliability.

The *reliability* of a system as a function of time, $R(t)$, is the conditional probability that the system has survived the interval $[0, t]$, given that the system was operational at time $t = 0$ [3]. In other words, the reliability is the probability that the sytem will provide correct output throughout a complete interval of time. A fault-tolerant system is reliable even after the occurrence of some number of *failures*. Failures that are present from the time of manufacture of initial assembly are often called *defects.* *Faults* are logical models (or abstractions) of the actual defects and failures that occur.

Phenomenological causes of faults are *physical* and *man-made.* Hardware faults are physical faults in the equipment that can be tolerated by introducing redundancy into the design. Replicating processors yields a straightforward and effective solution. Clearly, it is desirable to minimize the amount of resources dedicated to fault tolerance-related activities because this increases the resources available for more "productive" computing. This can be done through *selectivity.* Selectivity of fault tolerance should be in two forms: first, a choice as to whether to have any fault tolerance; second, if fault tolerance is required, then to use the resources judiciously.

In this dissertation, we first present an evolutionary look at some important techniques used in the design and implementation of fault-tolerant computing systems, paying particular attention to those systems that advocate selective fault tolerance. Then we propose an FPGA-base architecture that supports selective fault tolerance. In addition, we use the same basic mechanisms that we have added to support selective fault tolerance to also implement efficient multiprocessing.

1.1 Problem Formulation

Hardware fault-tolerant digital systems are currently designed with redundant computing modules so that a failure of a module does not mean failure of the system. Tradeoffs are made regarding the way the fault tolerance is implemented. Figure 1.1 identifies the main tradeoffs that designers consider.

	Benefits	Costs
Software Implementations	Flexible	Slow
Hardware Implementations	Fast	Inflexible

Figure 1.1: Fault Tolerance Implementation Tradeoffs

Software implementations allow the redundant resources to be used most efficiently, but considerable overhead is incurred in providing the fault tolerance. Hardware implementations offer the fastest solution, but their inflexibility in how the redundant resources are used is wasteful when applications do not need all the available modules to meet their reliability requirements.

Using traditional technology, designers of hardware fault-tolerant digital systems must deal with this dichotomy between hardware and software implementations in creating the system's fault-tolerant capability. Our goal therefore, is to *obtain a solution to the problem of fault tolerance that offers the benefits of hardware and software implementations, while mitigating the costs of both.*

1.1.1 Historical Perspective

Fault-tolerant digital systems strive to continue operation in the presence of faults. To tolerate hardware faults, additional hardware is introduced. If a copy of the redundant hardware fails due to a fault, then the fault-free copies can perform the operation. Additional logic is necessary to receive and resolve the outputs of the redundant copies to produce a single, fault-free output. How much redundancy is needed depends on the probability that a fault will occur, and on the system reliability or availability requirements. The most straightforward response is to introduce sufficient fault tolerance to handle the worst-case scenario for the system. However, during a system's mission the worst case may seldom occur, so the additional hardware for fault tolerance often does not serve a useful purpose. Researchers have recognized this, and, as a result, systems have been designed that switch between fault-tolerant mode and a mode where the otherwise redundant hardware is used for other functions. For example, the C.vmp (*V*oted *M*ulti*P*rocessor - circa 1975) [4] can switch between fault-tolerant mode and non-fault-tolerant mode. When in fault-tolerant mode the majority output from the redundant modules is computed by a hardware voter. Having only the two modes of operation, fault-tolerant and non-fault-tolerant, was not enough for all cases. During some missions, instead of using the full complement of redundant hardware, the reliability requirements could be met with a lesser amount of fault tolerance.

The need for finer granularity in choosing the requisite amount of redundancy led to the design of systems where hardware resources could be divided among concurrent tasks such that the redundancy would be judiciously allocated for a desired level of fault tolerance. In order to support this granularity it was necessary to have an easily modifiable voting operation that could be performed on some collection of the computing modules. Because of the flexibility required, this function has almost always been implemented in software. The SIFT (Software Implemented Fault Tolerance) Program [5] (circa 1975) was the first attempt at supporting fine-grained flexibility in fault-tolerant operation. This program also served as the seedbed for solutions to important problems in fault-tolerant distributed systems such as the Lamport's "Byzantine Generals Problem" [6]. In SIFT, the use of software-based voting permitted novel approaches for the tolerance of software errors (i.e., design errors). Software-based voting, however, produced a significant negative impact on performance.

A central theme of computer organization is that hardware and software are *logically* equivalent, and that the design decision to put certain functions in hardware and others in software is made on the basis of such factors as cost, speed, and expected frequency of design changes [7]. Reconfigurable devices such as Field-Programmable Gate Arrays (FPGAs) have altered this balance between hardware and software tradeoffs. These devices, which represent a revolution in custom digital logic design, consist of programmable logic cells and interconnections. FPGAs are becoming larger with even greater numbers of signal I/O pins. Recently, IBM announced a high I/O FPGA (448 signal I/O pins in a 624-pin Ball-Grid Array) using the Atmel FPGA architecture. Traditionally, only the most frequently-used operations had the cost justification for performing them in hardware; but now a single FPGA can be frequently reprogrammed in order to perform a variety of operations directly in hardware. The basis of this is the general rule that for any algorithm to achieve maximum throughput it should be implemented in hardware. FPGAs offer the opportunity to mend the split between hardware and software solutions to fault tolerance. Gaining the speed of hardware *and* the flexibility of software is now possible.

The objective of this research is to develop a capability for computer systems to achieve selective fault tolerance. With this capability, both parallel and fault-tolerant computing are supported. We investigate the use of FPGA technology to provide selective fault tolerance, and develop a design methodology based on this device technology. The FPGA is a hardware alternative for supporting flexible fine-grained hardware fault tolerance that has previously been implemented only in software. This permits significant speedup of the critical operations associated with fault-tolerant computing and provides a configurable switch for parallel computing.

1.2 Organization

The organization of this dissertation is as follows. In Chapter 2 background on some relevant topics and a literature survey of related work is presented. The requirements of high-performance multiprocessing and fault tolerance are covered in Chapter 3. In Chapter 4 we provide the architecture of the *Dynamic Reconfigurability Assisting Fault Tolerance (DRAFT)* system for providing FPGA-based selective fault tolerance. How DRAFT supports both multiprocessing and fault tolerance is presented in Chapter 5. We show the performance gains, cost, configurability, reliability, and scalability advantages of DRAFT over comparable designs in Chapter 6. In Chapter 7 we present implementation issues: modeling and simulating the FPGA-based system; applying dynamic reconfigurability; testing the FPGA by performing fault simulations on a logic model of the FPGA; and FPGA versus microprocessor reliability analyses. Chapter 8 summarizes and concludes this dissertation. We also discuss future work to extend the research presented in this thesis.

Chapter 2

Background and Literature Survey

2.1 Design Considerations for Fault-Tolerant Computers

Fault-tolerance is the ability of a system to continue executing its tasks despite the occurrence of some number of faults. Fault-tolerance can be achieved by *fault masking.* Masking (also called *passive*) redundancy is incorporated in the design to concurrently correct the errors caused by the occurrence of faults, and thus prevent the propagation of these errors to other modules. The most common example of passive redundancy is the *Triple Modular Redundant* (TMR) system. Another approach for providing fault-tolerance is *active redundancy*, which uses spare components to replace faulty modules once they are detected. A combination of these forms, called *hybrid redundancy*, applies passive and active redundancy to achieve fault-tolerance. In general, the design of a fault-tolerant computer involves using one or more of the following strategies [8]:

1. **Fault detection.** Hardware and software mechanisms are used to determine the occurrence of a fault. Fault detection mechanisms include: concurrent fault detection; stepwise comparison; and periodic testing to check if computers or communication links are operating correctly.

2. **Fault masking.** Concurrent masking and correction of generated errors.

3. **Fault containment.** This prevents propagation of erroneous or damaged information in the system after the occurrence of a fault, but before its detection.

4. **Fault diagnosis.** This locates and identifies the faulty module responsible for a detected error.

5. **Repair/reconfiguration.** This eliminates or replaces the faulty module, or provides the means to bypass it.

6. **Fault recovery.** This corrects the system to a state acceptable for continued operation.

Most of these techniques have been used to build fault-tolerant computers such as the Tandem non-stop architecture, Stratus pair and spare architecture, VAXft 3000, Teradata, Sequoia, FTMP and SIFT, and AT&T Electronic Switch System ESS [9, 10]. The effectiveness of fault-tolerance techniques can be measured by the fault recovery "coverage," which is defined as the conditional probability of recovering from a fault given that a fault has occurred [11]. It is difficult to measure this parameter because it involves evaluating the probability that fault detection, fault diagnosis, repair/reconfiguration, and recovery algorithms will operate correctly.

2.2 Fault Tolerance Methods

We now consider the implementation of hardware fault tolerance methods. In these implementations we focus on techniques that involve hardware redundancy; however we also mention three other types of redundancy that apply to fault-tolerant computing. These are types are *information redundancy*, *time redundancy*, and *software redundancy* [3].

The addition of redundant information to data to allow fault detection, fault masking, or possibly fault tolerance involves the use of codes. The simplest code is the parity code. Other codes include: m-of-n code; duplication codes, checksums, cyclic codes, arithmetic codes, Berger Codes, and Hamming Error-Correcting Codes. The applicability of a code depends upon the encoding and decoding process, whether error detection, error correction, or both are required, and the number of bit errors that need to be detected or corrected.

Time redundancy involves the repetition of computations so that faults may be detected or tolerated. Examples of methods that use time redundancy are: recomputing with shifted operands, recomputing with swapped operands, and recomputing with duplication with comparison.

Software redundancy methods may use extra, or redundant, software to detect and tolerate faults that occur in hardware. Tolerating faults within the software itself is through the use of functionally redundant, but differently designed software. These are design faults, which will be treated as a separate concern.

Because information redundancy can require large amounts of extra hardware [3], and time redundancy calls for significant costs in terms of throughput, we focus on hardware redundancy. Also, advances in processor technology have influenced our decision. For example, instruction retry has the attributes of time redundancy. When a transient fault is detected (through the use of time or information redundancy), execution must be rolled back to a state previous to the instruction for which the error occurred. Ideally, the fault detection is immediate and rollback requires that only the current instruction be re-executed; however, the deep pipelining and super scalar architectures of today's processors make this extremely difficult. As a result, using redundant processors provides a straightforward approach to tolerating faults (including transient faults).

Primary examples of fault tolerance methods where the digital output is produced by a hardware mechanism are found in [3], and are as follows:

N-Modular Redundancy (NMR) This represents passive hardware redundancy (see Figure 2.1). In most cases, N is selected as an odd number so that a majority voting arrangement can be used. If fewer than a majority of modules become faulty, then the remaining modules mask the results of the faulty modules. By comparing the majority with each module's output, a faulty module can be identified. When the number of faulty modules is greater than or equal to $N/2$, the correct majority cannot be determined. A common form of NMR is triple modular redundancy (TMR). We will refer to TMR as 3MR.

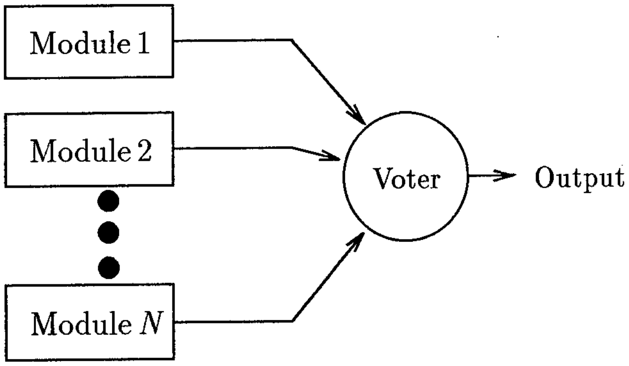


Figure 2.1: N-modular Redundancy

Duplication with Comparison (DWC) This is the fundamental form of active hardware redundancy (see Figure 2.2). In this method, two modules operate in parallel and their results are compared to provide a primary error detection capability. When a miscompare occurs, the result is ignored and module self-test or externally-applied diagnostic routines are used to identify the faulty module. Once identified, the faulty module can be replaced.

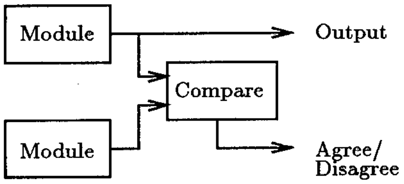


Figure 2.2: Duplication with Comparison

Standby Sparing This is a form of active hardware redundancy (see Figure 2.3). In this method, one module is operational and the other modules serve as spares. Various error detection schemes are used to determine when a module becomes faulty, and fault location is used to determine which module is faulty. A faulty module is removed from consideration and replaced with a spare. A switch examines the error reports from the error detection circuitry associated with each module to decide which module's output to use.

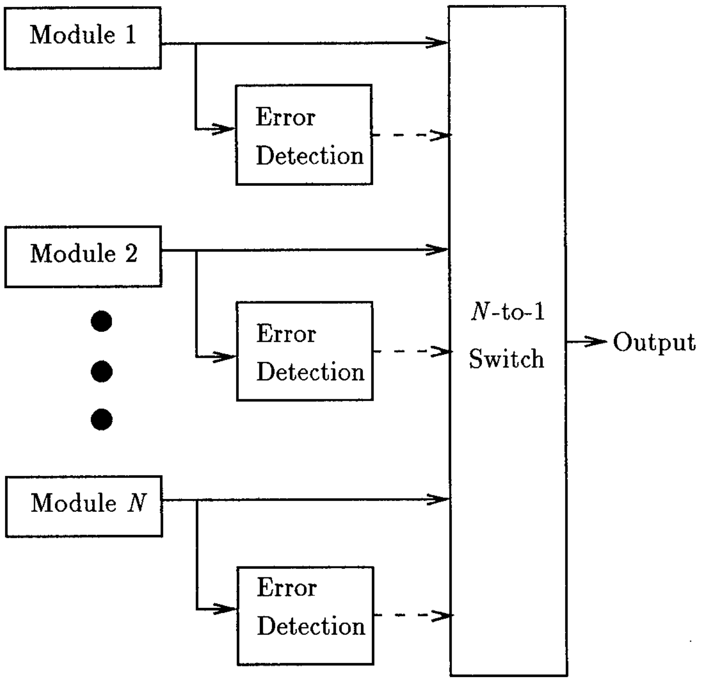


Figure 2.3: Standby Sparing

Pair-and-a-Spare This is another form of active hardware redundancy (see Figure 2.4). In essence, this method uses standby sparing; however, two error-free modules always operate in parallel and their results are compared to provide a primary error detection capability. When a miscompare occurs, the error reports from the modules are used to identify the faulty module and select a replacement module.

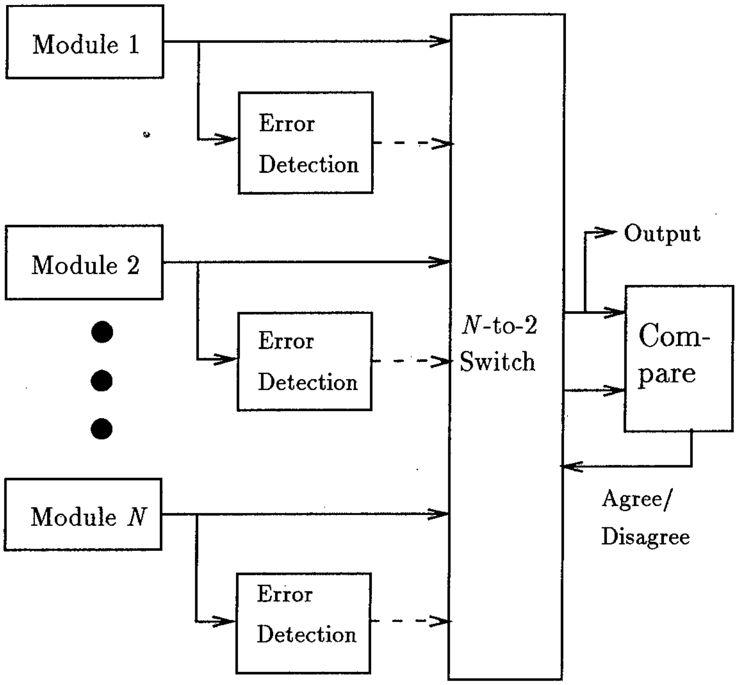


Figure 2.4: Pair-and-a-Spare

NMR with Spares This is form of hybrid redundancy (see Figure 2.5). This method provides a basic core of N modules arranged in a voting configuration. In addition, spares are provided to replace failed units in the NMR core.

Self-Purging Redundancy This is another form of hybrid hardware redundancy (see Figure 2.6). All modules actively participate in this method; any module can be removed from the system in the event that its output disagrees with the voted output of the system. A switch is associated with each module and the function of

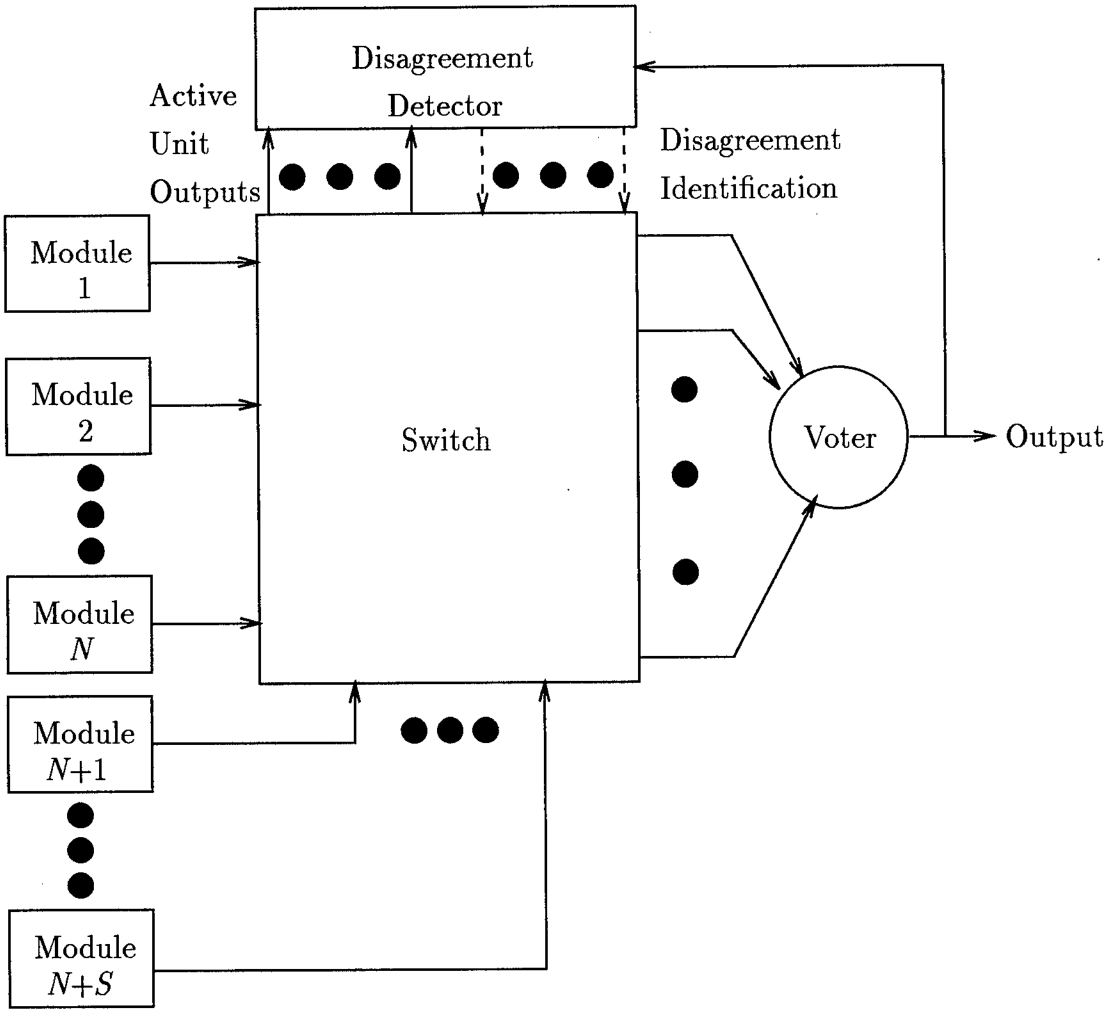


Figure 2.5: NMR with S Spares

the switch is to remove its module from the system in the event that the module fails. The voter that produces the system output and masks faults is a threshold voter; for the self-purging system of Figure 2.7, the effective threshold of the voter is fixed at half the number of the remaining fault-free modules. This has been shown to be the best self-purging system possible [12].

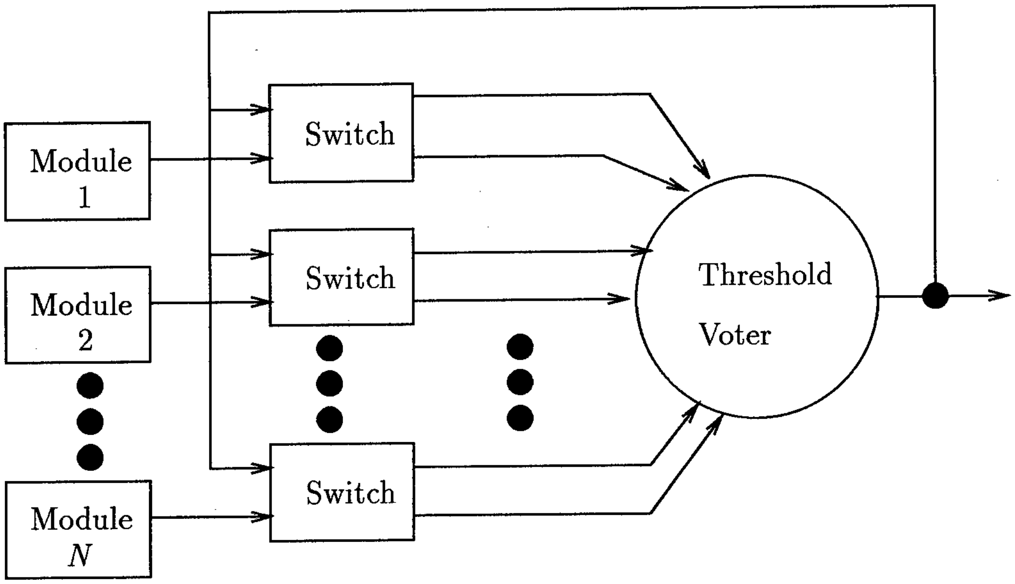


Figure 2.6: Self-Purging System for Optimal Tolerance to Multiple Faults

Sift-Out Modular Redundancy This is another form of hybrid hardware redundancy (see Figure 2.8). This method uses the N modules configured into a system using the circuits *comparator*, *detector*, and *collector*. The comparator compares each module's output with the remaining modules' outputs. Thus the comparator must produce one signal for each pairwise comparison that can be performed. The

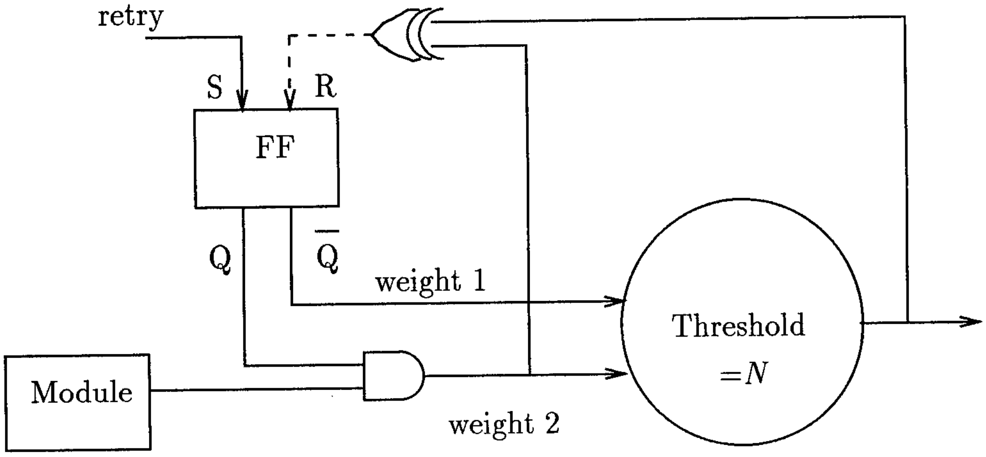


Figure 2.7: Switch for Optimal Fault Tolerance

total number of signals produced by the comparator is $\binom{N}{2}$. The detector produces one signal for each module for a total of N signals. The value of a signal is 1 if the module disagrees with a majority of the remaining modules, and it is 0 if the module agrees with a majority of the remaining modules. The collector uses the signals from the detector that indicate which modules are faulty and produces the system's output.

Another method, not listed in [3], is where a 5MR is automatically reconfigured into a 3MR system under single or double failures [13]. Since the majority function is symmetric, the 3MR is easily obtained from a 5MR by replacing any one input signal by 0 and any other input signal by 1. The reconfiguration is accomplished in such a way that when the 5MR detects a malfunctioning module the faulty unit is suppressed and a fault-free module is removed and effectively held in reserve. In this way, the 5MR

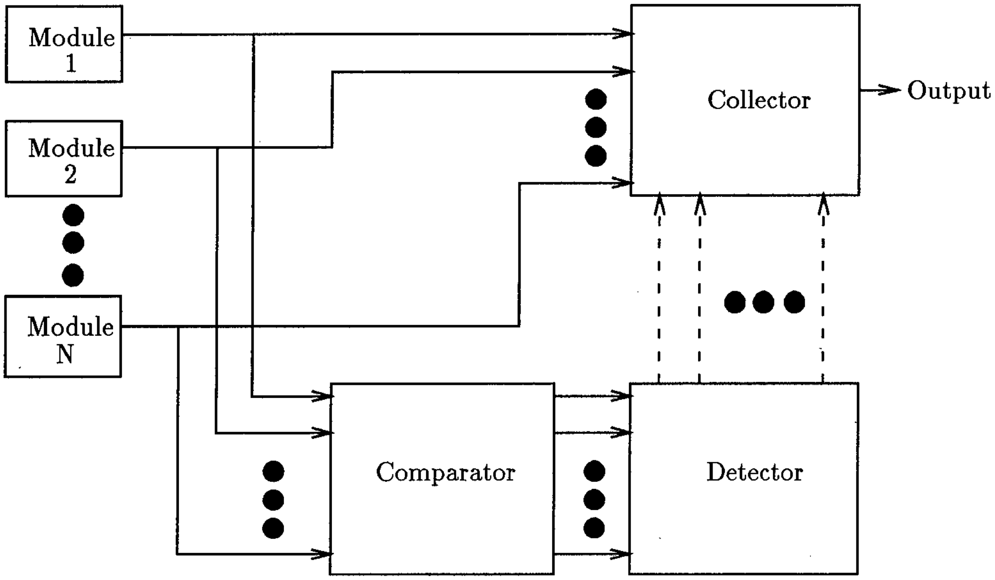


Figure 2.8: Sift-Out Modular Redundancy

becomes a 3MR with a spare module. After two modules have failed, the system is still a perfect 3MR. This method can tolerate more failures than 5MR can, and has a circuit realization that is simpler than the other methods that employ hybrid redundancy.

A method proposed by [14], called *Stepwise Negotiating Voting*, uses error reports from the modules in choosing the proper system output. As the number of non-faulty modules decrease, this method uses a switch matix to gracefully degrade (negotiate) the inputs to a conventional voter. A system composed of N modules can tolerate faults in up to $N-1$ of the modules.

2.3 Switching Between Fault Tolerance and Non-Fault Tolerance

When protection from faults is unnecessary, redundancy is a waste of resources. For instance, during non-critical tasks the modules should be scheduled with nonredundant (i.e., distinct) programs. Since not all tasks require fault-tolerance, it would be desirable to run only the critical operations in a fault-tolerant mode, and run the rest of the operations in a normal mode. This would provide a significant improvement in performance without compromising the fault-tolerance requirements. Consequently, the architecture of the computing modules should support dynamic reconfigurability such that the processors within a node can be configured to be used as a masking redundancy during critical operations and as a multiprocessor system during non-critical operations. This capability has been supported by the C.vmp (computer voted-multiprocessor) which contains three processor-memory pairs that can operate independently as will as provide fault-tolerant operations [4].

Figure 2.9 shows the basic configuration of the C.vmp. Figure 2.10 shows the C.vmp voter/multiplexing architecture. PA, PB, and PC are the buses for processor A, B, and C respectively. V is the voting element. L is a latch for ensuring address timing integrity. EA, EB, and EC are the external buses A, B, and C respectively. The voter is used to determine the modes of C.vmp operation. In fault-tolerant mode, the transmitting portions of the three buses are routed into the voter, and the results are routed to the receiving portions of the three buses. In non-fault-tolerant mode, buses B

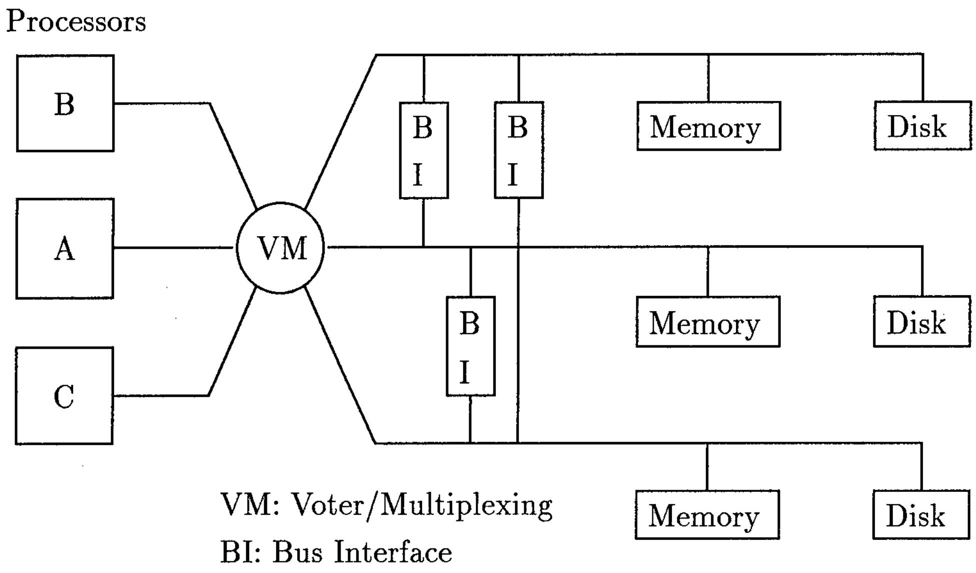


Figure 2.9: C.vmp Configuration

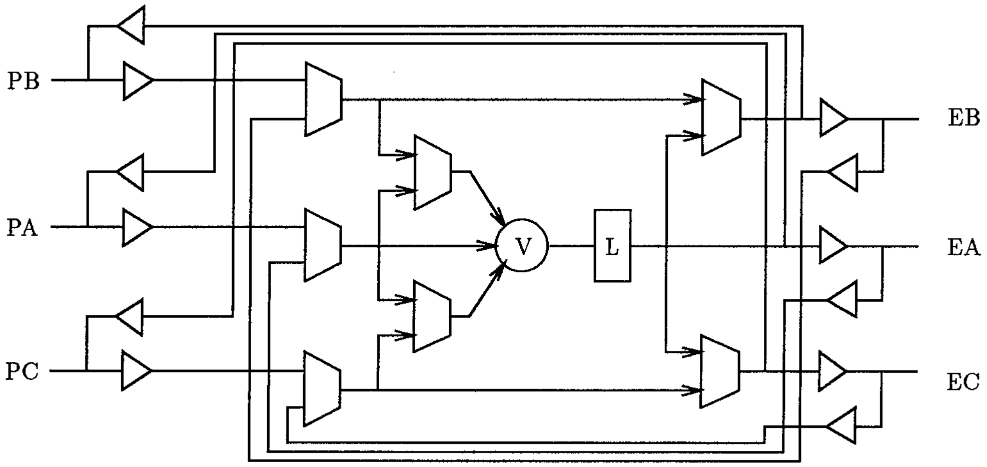


Figure 2.10: C.vmp Voter/Multiplexing

and C are routed around the voting hardware while bus A is routed to feed its signals to all three inputs of the voting element. In this way, the C.vmp forms a loosely-coupled multiprocessor.

2.4 Selective Fault Tolerance

Selective fault tolerance supports a *fault-tolerant mode*, *parallel computing mode*, or both. Under selective fault tolerance, the computing resources are apportioned so that they are judiciously used. For instance, a task in fault-tolerant mode that does not require the full complement of redundant hardware has assigned to it only the necessary amount of resources. By making the remaining resources available to other tasks, concurrent task execution can be accommodated.

As shown in Section 2.2 there are several methods that receive and resolve the outputs of the redundant tasks to produce a single, fault-free output. We refer to this operation, regardless of the method used, as *fault decision.*

Selective fault tolerance implies flexibility in the fault decision. Software takes advantage of a processor's computational capability in yielding flexibility. In NMR, the output majority can be determined by a software voter, and, as failures in the redundant modules occur (and accumulate), the software can easily accommodate the fault decision of fewer than N modules. By simply modifying the software, the manner of voting can be carried out differently with respect to changes in the system. However, a disadvantage is that the processor cannot execute instructions and process data as rapidly as a dedicated hardware voter [3].

In the 1970's two fault-tolerant systems were developed that took opposite approaches to voting. The Fault-Tolerant Multiprocessor (FTMP) took a hardware approach and the Software Implemented Fault Tolerance (SIFT) System used software voting. In FTMP [15], a set of processors and memories are connected to five redundant buses through special redundant bus "guardian" circuits. Processors and memory modules are dynamically assigned to be a member of a group of three processors and three memories which run the same task. This is designated as a *triad.* The guardian circuits in the processors vote on the three copies of the data arriving from their assigned memories, and, conversely, the guardian units in the memories vote on the data from their processors. The occurrence of a failure can be sensed by a different triad, and the affected triad can be reconfigured by assigning a new processor, memory, or bus to the triad that experienced the failure. The computers in SIFT [16] are totally connected. To perform a vote, the processors send their results to the other processors where a software voting procedure is invoked to mask faults. As noted in [17], an advantage of FTMP over SIFT is that the voting is done in hardware whereas SIFT spends a significant percentage of time running the software voting program. However, as also noted in [17], although FTMP may have been more practical than SIFT, the software approach made the more significant research contribution due, in part, to software voting allowing tasks to run in fault-tolerant and non-fault-tolerant mode. In retrospect, the SIFT designers (who sought to emphasize software implementations of the system functions), cited the overhead for software voting as putting a heavy damper on performance, and stated that the use of a hardware voter would have been "...a profitable retreat from purity of concept" [5].

Two multiprocessors designed in the early 1980's targeted selective fault tolerance [18] [19]. In [18], the term *taskspecific* redundancy is used to capture the notion of selective fault tolerance. Depending on the application, very different reliability requirements can be met by this fault-tolerant computer system. These requirements define the fault tolerance methods that the system provides and their implementation strategies. The two fault tolerance methods provided are 3MR and duplication with comparison. When no fault tolerance is necessary, the task requires no redundancy. Each task is individually equipped with the suitable degree of fault tolerance and therefore tasks with different redundancies are mixed within one system. Since these tasks may be implemented in the system in any combination a high degree of flexibility for the voting and comparing process is required, and, as noted in [18], *only a software based voting mechanism seems possible.* The ATTEMPTO system [19] implements taskspecific fault tolerance. The system enables the user to choose an appropriate balance of trade-offs in terms of throughput and fault tolerance based on the user's application. The user can direct that part of the system resources be spent on reliability purposes with a resultant reduction of system throughput. When the resources are used exclusively for maximizing system performance, then reliability is sacrificed. Since the system is built from conventional computer components, its fault-tolerance mechanisms required software implementations.

In the late 1980's the Multicomputer Architecture for Fault Tolerance (MAFT) was proposed as a distributed system that relies on modular redundancy and distributed agreement algorithms to mask the effects of corrupted messages [20]. The modules are connected using a fully connected broadcast network. Each data message is tagged with a *data identification descriptor (DID)* that uniquely labels the data contained in the message. The voter in each module does not wait for the arrival of all expected copies of a DID. Rather, it performs a vote using the new copy and any previously received copies. If one copy of the task generating the data value should fail such that it ceases producing messages, the voted value of the remaining is still available. The amount of redundancy assigned to a task is variable; MAFT therefore supports selective fault tolerance. When changes in the system operating set occur due to module failures, the process of task reconfiguration redistributes the application workload. Additional flexibility is designed into the voter to allow diversity in the modules' software and hardware. Design diversity is used to tolerate design faults. The flexibility of the MAFT is achieved largely because the algorithms for fault tolerance are implemented in software.

The design of fault-tolerant distributed systems has been discussed at length in [21]. In [21], a computing service specifies a collection of operations that can be carried out only by a server for that service. A hierarchy exists in the architecture of the computer system where servers implement their service, in turn, through services implemented by other servers. What is a *server* at a one level of system abstraction can be a *user* of the service at another level of abstraction. *Failure semantics* is a term used for characterizing behaviors in the presence of failures. *Weak failure* semantics imply that there are more ways for a server to fail, and *strong failure* semantics means that fewer failure semantics can be experienced with a server. Tradeoffs must be made when deciding on a server's failure semantics:

> ...a key issue in designing multilayered fault-tolerant systems is how to *balance* the amounts of failure detection, recovery, and masking redundancy used at the various levels of a system in order to obtain the best possible *overall* cost/performance/dependability results.

[21]

Investing resources at a lower level of abstraction to ensure that lower-level servers have a strong failure semantics can contribute to cost savings and speed improvements at higher levels of abstraction. Conversely, deciding to use too much redundancy at the lower levels of system abstraction might be wasteful from cost/effectiveness point of view, since such low-level redundancy can duplicate the redundancy that higher levels of abstraction must use to satisfy their own dependability requirements.

To achieve an appropriate balance at the low levels of system abstraction, there must be freedom to choose how much redundancy to employ. Selective fault tolerance provides this necessary freedom; however, we have discussed that, in today's state-of-the-art, *software* implements the fault decision. In this dissertation, FPGA-based selective fault tolerance offers the required flexibility of the state-of-the-art, but with the significant performance gains of a *hardware-based* approach.

Chapter 3

Requirements for High Performance Multiprocessing and Fault Tolerance

In this chapter we discuss the requirements for high-performance multiprocessing and fault tolerance. The design must allow a variable number of processors per application, and for fault tolerance both passive and active hardware redundancy must be allowed. As noted in Section 2.2, NMR represents passive redundancy, and duplication with comparison (DWC) is a fundamental form of active redundancy. It must be possible to apply these fault tolerance methods to any combination of the computing modules: passive redundancy among any n ($n \leq N$) modules, or active redundancy between any two modules. The apportionment (by the system) of passive redundancy to any n modules means direct availability of hybrid redundancy.

Multiprocessing is required. In this case, the modules that are not dedicated to providing fault tolerance can be used to provide concurrent execution of non-redundant computations. Parts of a single application can therefore be carried out in parallel.

Because provisions for both fault tolerance and multiprocessing are required, we are specifying *selective fault tolerance* capability.

3.1 Multiprocessing Support

An application may be composed of several tasks. One approach to represent applications is by graphs where nodes represent computational tasks, and arcs represent dependency constraints and/or communication flow between the tasks. Figure 3.1 shows a diagram of an application with five tasks. For multiprocessing, there are two main parallel processing programming paradigms: *functional parallel* and *data parallel.* Figure 3.2 shows the graph of an application programmed according to the functional parallel model, and Figure 3.3 shows the graph of an application that conforms to the data parallel model. From these application graphs, tasks are assigned to individual processors. The applications are compiled using the processor identifiers shown in the application graphs; however, these are *virtual* ids and, as will be discussed later, must be mapped to physical processors at runtime. For either parallel programming model, we say that the tasks that comprise the multiprocessing application are in the *MP* mode.

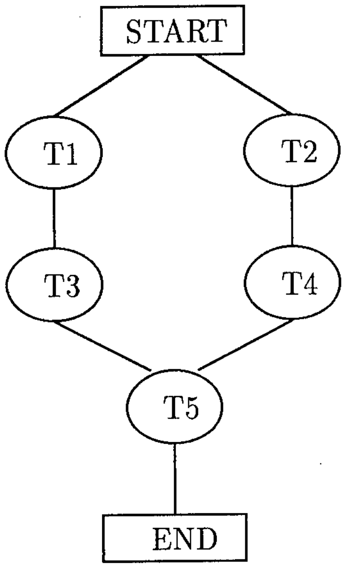


Figure 3.1: A Diagram of an Application with Five Tasks

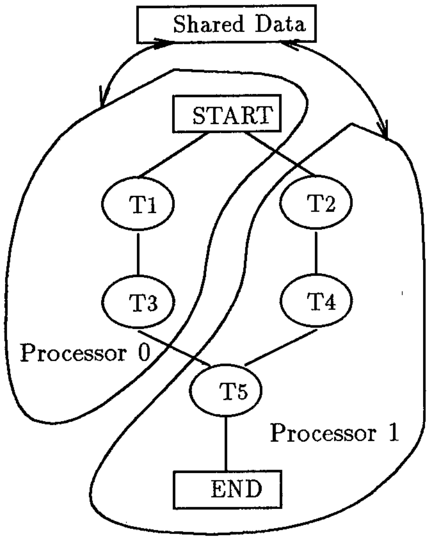


Figure 3.2: Functional Parallel Model

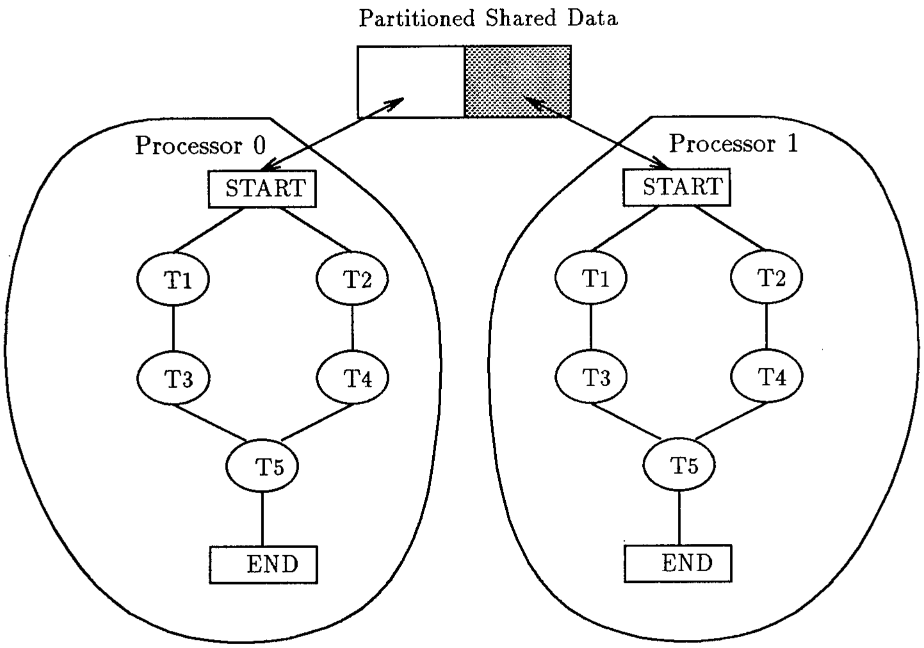


Figure 3.3: Data Parallel Model

3.2 Fault Tolerance Support

Fault-tolerant applications require multiple processors to execute the same task. The number of processors depends on the number of faults to be tolerated. The type of redundancy (active or passive) used depends on whether the effects of the faults must be masked or not. A fault-tolerant application and its associated tasks are said to be in the *FT* mode. Figure 3.4 shows a graph of an application that must mask the effect of a faulty processor, so three processors are used in a 3MR fashion to form a fault-tolerant computer. In this graph, each of the five tasks are assigned to the three redundant processors, and the final output of the application is the result of a majority vote. The transition to an application with combined FT and MP requirements is easily achieved by replicating the processors that form a multiprocessor.

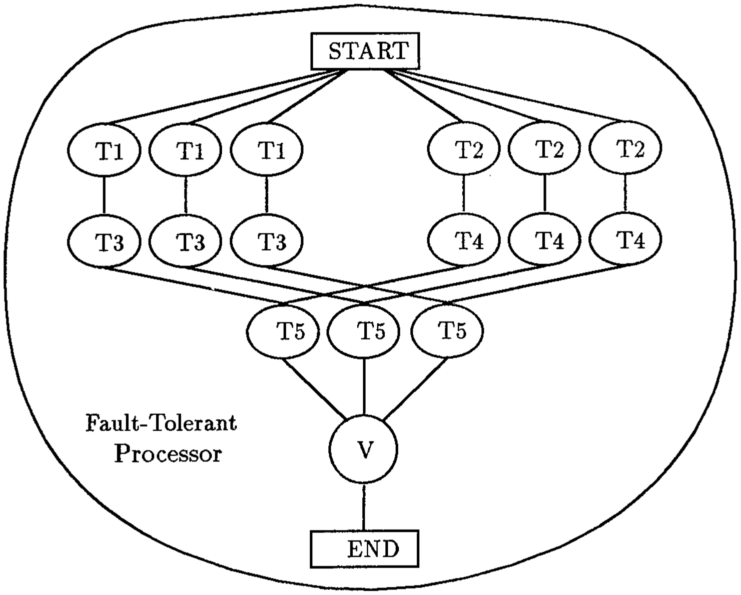


Figure 3.4: Fault-Tolerant Application

3.2.1 Tolerating Software Faults

In addition to the well-established fault tolerance techniques for coping with the occurrence and effects of anticipated hardware component failures, it may be necessary for an application to also be able to tolerate its own *design* faults. While not considered a major problem in hardware, design faults are a major concern in software [22]. Software is a critical component of all electronic computing systems. Indeed, the major portion of a system's complexity is to be found in the software. Two main techniques have been proposed for tolerating software faults: the *Recovery Block (RB)* scheme [23] and *N Version Programming (N VP)* [24]. In general, these schemes employ multiple versions, or variants, of the software in an attempt to guarantee that at least one version will pass the correctness checks that are performed while the software executes. The checks include voting on results that may not exactly agree, but are nonetheless correct, and determining the reasonableness of a variant's results. In a study by Laprie *et al.* [25], RB and NVP were both recommended for meeting the requirements of a combined hardware-software fault-tolerant architecture where the checks that detect software faults are deemed sufficient to detect hardware faults as well. However, quantified estimates of the improvement in system reliability to be expected from using RB or NVP are lacking. As a result, these techniques are not widely accepted (as discussed by Lee and Anderson [22]). NMR and duplication-with-comparison together remain the cornerstone of hardware fault tolerance.

An architecture for high-performance fault-tolerant computing must provide the hardware capacity to rapidly and selectively vote or compare processor outputs. Although current usage of software fault tolerance is limited to only a few systems, its potential as a source of additional system dependability has been demonstrated [28], so our proposed architecture must support it. The architectural capacity that enhances hardware fault tolerance has to also be extensible to software fault tolerance.

Already, software solutions to hardware fault tolerance been found to severely curtail performance [26] [27]. Techniques for tolerating hardware faults are well-established, yet their software implementations have recognized and serious shortcomings. Software fault tolerance requires that checks be performed, and these checks are far more complex than simple output comparison or majority voting. It is therefore inevitable that, to tolerate software faults effectively, the required checking operations must be carried out in hardware.

3.2.2 Process Recovery for Resource Preservation

Not all hardware faults are permanent; one study found that from 60 to 80 percent of hardware faults were temporary [29]. A transient fault is a fault that is present in a system for only a limited period of time. A fault that is transient should not be treated as a permanent fault: the occurrence of a permanent fault means the module in which is occurred must be replaced, but a transient fault spontaneously disappears and the module is once again usable. Restoration of the system is accomplished by replacing the current erroneous state with a good state it occupied before the fault occurred. *Checkpointing* is a process where periodic "snapshots" of the system's state are taken so that, if a subsequent fault is transient, then the state prior to the fault can be recovered.

The architecture therefore must include support for detecting errors in intermediate results to allow recovery from transient faults. To adequately checkpoint a process, in principle a copy must be made of main storage as well as any secondary storage accessible to the process [22]. Potentially, the amount of data involved is enormous, and to ensure the integrity of the checkpoint all these data must be examined for latent errors. Thus, checking of the checkpoint data is required, and the only conceivable way this can be performed efficiently is through a high-speed physical architectural mechanism.

3.3 Operation Methodology

Applications are assumed to arrive from a host that requests application service. As mentioned earlier, there must be two operational modes: fault-tolerant (FT), and multiprocessing (MP). The host describes an application to the server by specifying the required operational mode (FT or MP), the amount of computing resources needed by the application, and the distribution of the application's tasks among those resources. All this information is conveyed by the application graph. To illustrate the type of support that must be provided, consider two multiprocessing applications APP1 and APP2, and fault-tolerant application APP3. APP1 is programmed according to the functional parallel model and is described by the application graph of Figure 3.2. APP2 is programmed according to the data parallel model and conforms to the graph of Figure 3.3. APP3 follows the graph of Figure 3.4. For this example, we assume that five processors are available and the application scenario is as follows:

1. APP1 and APP2 arrive.
2. APP1 and APP2 are loaded and execution begins.
3. APP1 completes and APP3 arrives.
4. APP3 is loaded and execution begins.
5. APP2 and APP3 complete.

In this scenario, the following steps are performed:

- For APP1:
  1. Load tasks T1 and T3 into processor 1, and tasks T2, T4, and T5 into processor 2.
  2. Initialize APP1.
  3. Signal processors 1 and 2 to execute their tasks.
  4. Coordinate communication (i.e., sharing of data) between processors 1 and 2.
  5. Produce APP1 results from processor 2.
- For APP2:
  1. Load tasks T1, T2, T3, T4, and T5 into processors 3 and 4.
  2. Initialize APP2.
  3. Signal processors 3 and 4 to execute their tasks.
  4. Produce APP2 results from processors 3 and 4.

APP1 completes.

- For APP3:

  1. Load tasks T1, T2, T3, T4, and T5 into processors 1, 2 and 5.
  2. Initialize APP3.
  3. Signal processors 1, 2, and 5 to execute their tasks.
  4. Coordinate the 3MR voting algorithm.
  5. Produce APP3 results that are the majority output from processors 1, 2, and 5.

In situations other than the above example, applications may be of different modes, arrive in different order, and require a different number of processors. Because we do not restrict ourselves to a particular situation, the functions described above must be fully flexible. Fault tolerance involves multiple processes, so it initially appears as just more parallel tasks. However, the output of parallel tasks across processor boundaries receive different treatment depending upon the application's mode. Furthermore, support for multiprocessing and fault tolerance should be hardware-based for maximum performance.

Chapter 4

Dynamic Reconfigurability Assisting Fault Tolerance (DRAFT) Architecture

Changes made to the internal logic of computer-based systems to enhance performance requires consideration of the architectural design. Previously, only the most frequently used operations had the cost justification for performing them in hardware, but now reconfigurable logic can be frequently changed in order to perform a variety of operations directly in hardware. Acting as a coprocessor, reconfigurable logic increases a processor's throughput; however, it does not parallelize execution as in a multiprocessor. Moreover, when multiple computing modules operate redundantly, they offer fault tolerance. The architecture presented in this chapter uses dynamic reconfigurability to support fault tolerance and multiprocessing. We refer to our architecture as *Dynamic Reconfigurability*

Assisting Fault Tolerance (DRAFT).

4.1 Architectural Description

The main components of DRAFT include a set of *computing modules (CMs)* (each composed of a processing unit and memory), dual-ported *Result Memories (RMs)*. a *Reconfigurable Logic and Switching Unit (RLSU)*, an *RLSU controller*, and read-only memory (ROM). This architecture is shown in Figure 4.1. The RLSU consists of a *Multimode Translator (MT)* and a *Main Controller (MC)*.

Each CM has a private RM in which it can place application output. Communication between the MT and the CMs is through the dual-ported RMs. In a similar way, the CMs and the MC communicate, via the MT, using the dual-ported RMs. The host communicates with DRAFT through the host interface unit.

The MT translates CM output for use in either FT or MP mode. That is, among its functions are voting between any CMs and linking any two CMs for message passing. The MT detection signals have different purposes, dependent upon the application's mode. In FT mode, they can indicate a failed CM, and for MP mode they can flag which CM is the source of an output message. The MC coordinates the activities between the MT, RM, RLSU controller, and the Interface Unit.

Through the use of each CM's private RM, MT service to one application does not force suspension of the output activities of any concurrent application; instead, output is temporarily buffered for MT service. With *double-buffering*, RM read and write operations can be performed in parallel. By dividing the RM into two buffers, results

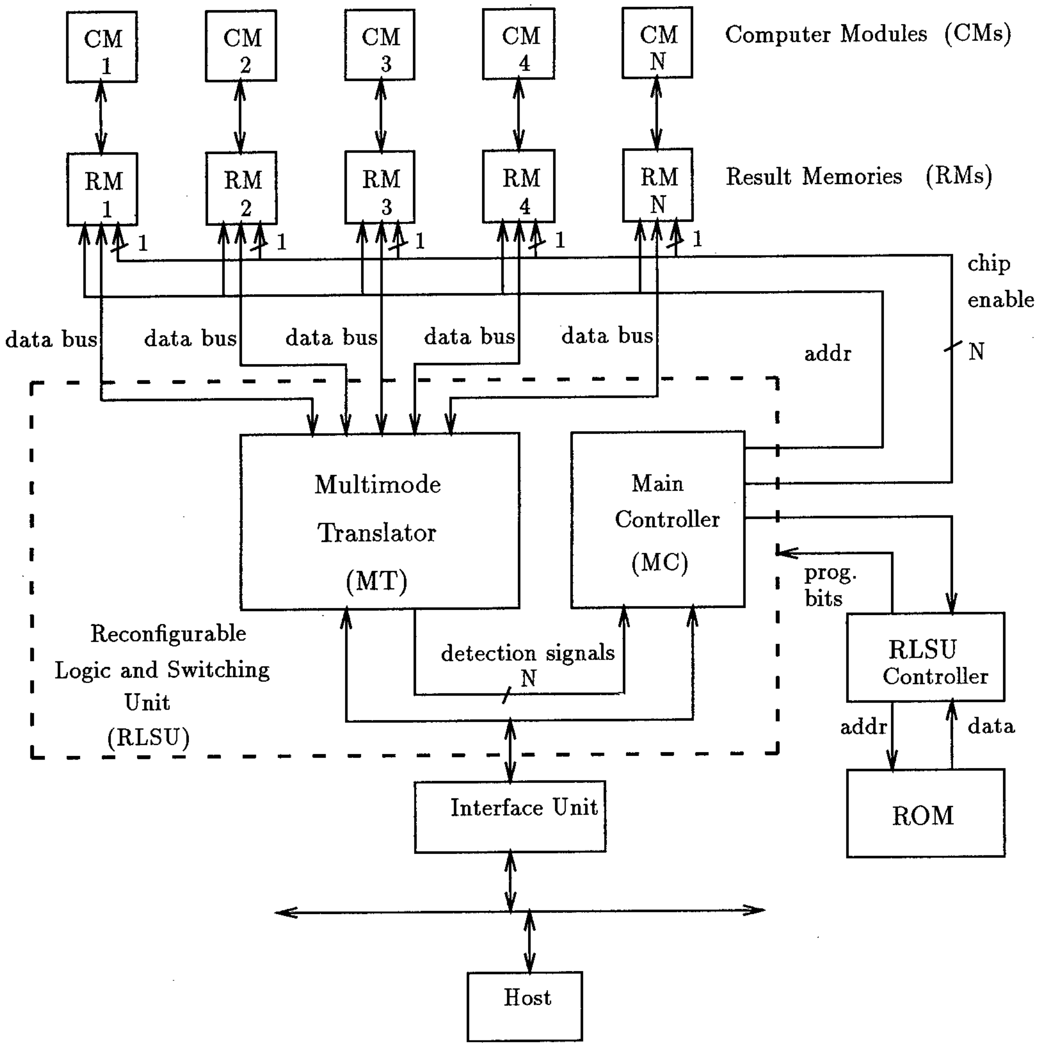


Figure 4.1: DRAFT Architecture

can be read from one buffer while results are written to the other buffer. Once the data from the first buffer have been read, the CM can then write to this buffer allowing concurrent reading of the second buffer. The two buffers continue to switch back and forth from *destination* to *source* of results.

4.1.1 Reconfigurable Logic and Switching Unit (RLSU)

The RLSU consists of a two-dimensional array of programmable logic blocks (or cells) that can be connected through interconnection resources (or buses). Figure 4.2 shows the basic makeup of an RLSU entity.

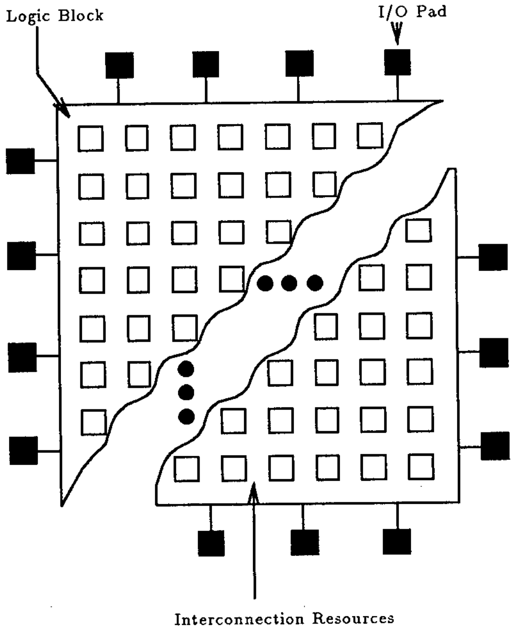


Figure 4.2: The RLSU Entity

The cells can implement a simple combinational logic function, a constant value ('0' or '1'), or a sequential function (i.e., a flip-flop). Inputs are provided by the cell's neighboring cells or from the adjacent buses that run horizontally and vertically through the inter-cell channels. A cell's output is to the neighboring cells and can also be placed on the buses.

The cells are individually programmed and reprogrammed to perform a logic function. Cell program data are stored external to the RLSU in non-volatile memory such as ROM. Cell programming takes place without disturbing the programming of any other cell, so that, once its programming is complete, a cell's logic function is unchanged by the programming of other cells. This is called dynamic reconfiguration, and, as a result, the RLSU can operate even while some of its cells are being programmed. Figure 4.3 shows how the RLSU could be used to implement multiple functions. In this example, one collection of cells implements function f_i, while a different collection of cells implements f_j. As only one function is needed at a time, the cells used by the other function are inactive so loading of that function's program data occurs without interruption to the operating function.

Because all cells are reprogrammable, the RLSU can be used repeatedly to perform many different functions. By clustering repeatedly-used building blocks of pre-designed functions, these groups of cells make up *virtual ICs (VICs).* The RLSU can be envisioned as a "circuit" board with sockets for accepting VICs. Insertion of different VICs, and personalization of the interconnect between these VICs, yield different board operations. DRAFT's operations exhibit a locality of reference, so that the need for future operations are anticipated and requests for their support are issued in advance. Handling requests for board operations require fetching the configurations for the necessary VICs from a

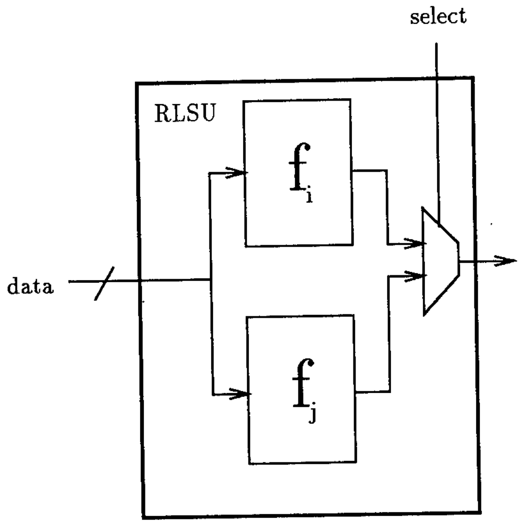


Figure 4.3: RLSU Dynamic Reconfiguration

library (stored in the ROM), and then using them to populate the board. A new VIC is plugged into a socket that is already occupied by simply writing over the cells of the unused, resident VIC. The functions that comprise an operation are never active all at the same time, so the operation can begin with only a subset of its functions actually loaded. Because modification of the board occurs dynamically, functions that are no longer needed are replaced during the functional latency. Control of the modification process is necessary to ensure that placement of a new function leaves a currently active function undisturbed.

The control process itself uses dynamic reconfiguration. Figure 4.4 shows the partitioning of MC functions into simpler functions. At boot up, the initial controller function (fcn 0) is configured by loading a counter with a 0 count. This is the starting address for fcn 0 stored in ROM. As the count proceeds, fcn 0 is loaded into the RLSU and then the count is halted by this function. All requests for RLSU functions are made through fcn 0. As shown in Figure 4.4, a request for fcn 13 is made to fcn 0. The starting ROM address is loaded into the counter.

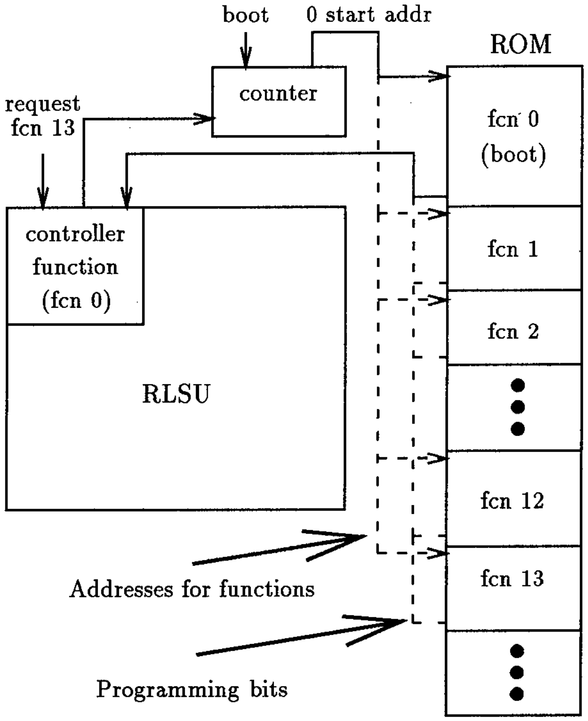


Figure 4.4: Controlling Dynamic Reconfigurability

Loading of fcn 0 is under control of the counter where the address bits used for accessing the ROM also serve to address cells for RLSU programming. A binary code stored in the ROM indicates the end of function (EOF). The controller function fcn 0 halts the counter once it recognizes that an EOF has been read. After boot-up, fcn 0 receives all function requests and coordinates dynamic reconfiguration of the RLSU. The RLSU controller shown in Figure 4.1 has the complexity of a simple counter.

Two types of cell addressing are used by fcn 0 to load a function into the RLSU: either the counter bits serve as start addresses and counts for block programming many successive RLSU cells, or the address of a single cell to be programmed is stored in the ROM along with the cell's programming data. Functions are numbered according to which type of addressing they require. When a function request is made, inspection of the function number by fcn 0 determines how the programming data are fetched from ROM and loaded into the RLSU. In support of dynamic reconfiguration, fcn 0 offsets the target cell address to steer it to an inactive area of the RLSU. In either type of cell addressing, function loading is completed when fcn 0 detects an EOF. Obviously, the EOF code cannot be in the code space for the cells' programming data or address.

As the coordinator for the dynamic reconfiguration process, fcn 0 is never overwritten by another function. This is analogous to the software that implements a page-replacement algorithm in a main-memory management scheme: it never allows itself to be paged out to disk because it would have no way to return.

4.1.2 Reconfiguration Methodology

For FT support, the RLSU provides voting on 5, 4, or 3 of its inputs, or comparison of any two of its inputs. A common operation in support of MP modes is the passing of messages between any two CMs.

Configuring the RLSU for both types of operations is accomplished by changing only a few cells. Through partial reconfiguration, the device's function changes while its structure is maintained, so that the time required to switch between the various operations is kept small. The implementation details are presented in Chapter 7; however, we describe here the methodology as it relates to the DRAFT architecture.

All the CMs have physical connections to the RLSU; yet an application that is not executing on all the modules uses only some subset of these connections. It is important to remember that the other connections still have an electrical value on them, and if these values are used, then an erroneous output could result. Ignoring these inputs is critical. For FT applications, the RLSU is programmed so that the majority function, expressed as a sum-of-products, implemented in a logic network can be reduced to the majority function of fewer inputs. Su and DuCasse [13] showed that, since the majority function is symmetric, a 3MR is easily obtained from a 5MR by replacing any one variable by 0 and any one other variable by a 1. With the 5MR voter, some cells are programmed to to a constant 1 or 0 in order to implement a smaller function by only "programming" gate elements; that is, there is no need for steering logic, new elements, or change in interconnect. When converting from a 5MR to a 4MR, the unused variable is replaced by a 0, which eliminates all the product terms with that variable. The remaining terms express the majority function of the four variables. In the RLSU, if the vote involves fewer than five inputs, then some cells are programmed to disable the detection signals of those CMs that are not participating in the vote. For DWC, some cells in the 5MR structure are programmed to route the outputs from the two participating CMs to the voter's disagreement detector for the comparison operation. One of these CMs results also serve as the primary RLSU output. Routing is similarly performed in passing a CM output as an interprocessor message. From the RLSU configuration for 5MR, support for other FT operations and MP are derived. Figure 4.5 shows the relationship between the RLSU configurations as a rooted tree, with 5MR as the root and the other configurations as the leaves.

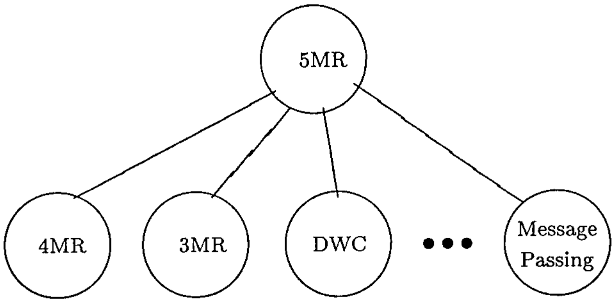


Figure 4.5: Rooted Tree Representation of Reconfiguration Methodology

To attempt to minimize both reconfiguration time and the number of distinct configurations that must be stored, our approach is based on incremental programming (or "deltas") to transition between standard configurations. For example, to switch from 3MR to DWC, 3MR is first reconfigured to the predecessor 5MR. From there, the 5MR is reconfigured to the successor DWC. During operation of DRAFT, when there is uncertainty about what function will be needed next the default RLSU configuration is for 5MR. In this way, the other configurations can be reached in minimum time.

Chapter 5

Multiprocessing/Fault Tolerance Support in DRAFT

In this chapter we present the procedures DRAFT performs, and detail the organizational features of the architecture. By taking a top-down approach, our initial view of DRAFT is that of the application developer. Our description progresses to how applications request and receive support for the MP and FT modes.

5.1 Top-Level View of DRAFT

At the top level, DRAFT appears to the host as a pool of processors that can be apportioned for concurrent MP or FT applications. During DRAFT operation, the MC first checks if there is a message from the host. If no host message currently exists and applications have been dispatched, then the MC polls the RMs for task results.

Figure 5.1 shows the MC control algorithm.

```
Forever do /*periodically*/
  if message_from_host_exists then
    respond_to_host
  else
    if applications_are_dispatched = true then
      poll_RMs_for_task_output;
end.
```

Figure 5.1: MC Control Algorithm

Next we describe the two branches of the MC control algorithm. That is, we will first describe how the MC responds to a host message, and then we describe the procedures for polling for task output.

5.2 Handling of Host Messages

The primary messages sent to DRAFT from the host are a request to run an application or *notification* that an application is complete. In the former case, if a sufficient number of CMs are available, then they are allocated and the application is accepted. In the latter case, the host notifies the MC that the application is finished, and the CMs allocated to the application are freed. Figure 5.2 shows the algorithm used by the MC to handle messages from the host.

5.3 Maintaining Application Information

The MC maintains a table, called the allocation table, for tracking where each application has been placed, and whether the applications are in the MP or FT mode. The

```
Procedure respond_to_host;
  if msg = "load $App_i$" then
    begin
      if num_CMs_needed($App_i$) > num_CMs_available then
        send(in : host, "cannot service $App_i$")
      else
        begin
          accept(in : $App_i$);
          allocate_CMs(in : $App_i$);
          num_CMs_available = num_CMs_available - num_CMs_needed($App_i$);
          dispatch_tasks(in : $App_i$);
          applications_are_dispatched = true;
          initialize(in : $App_i$);
          execute(in : $App_i$);
        end;
    end;
  if msg = "$App_i$ complete" then
    begin
      deallocate_CMs(in : App_i);
      if num_allocated_CMs = 0 then
        applications_are_dispatched = false;
      num_CMs_available = num_CMs_available + num_CMs_needed($App_i$)
                                - num_CMs_failed($App_i$);
    end;
  else
    begin
      msg = function(msg);
      send(in : host, msg);
    end;
end respond_to_host.
```

Figure 5.2: MC Host-Message Handling Algorithm

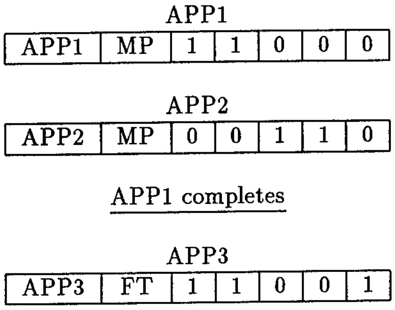
allocation table describes each application initiated. Figure 5.3 shows the format for a table entry.

Application id	FT/MP	b1	b2	b3	•••	bN

Figure 5.3: Allocation Table Entry Format

Included in an allocation table entry is an N-bit vector, called the allocation vector, that shows the assignment of CMs to an application, the applications' ids, and the applications' modes (fault-tolerant (FT) or multiprocessing (MP)). A '1' in the ith bit of the allocation vector means that the ith CM has been allocated to that application. In FT mode, the multiple '1's in the allocation vector show where the application's tasks have been redundantly loaded. The application graphs (see Figures 3.2 and 3.3) dictate the number of processors needed in MP mode. At compile time, the numerical id's given to the processors in the application graph are used for interprocessor communication. We assume here, without loss of generality, that processor ids in every application graph begin with number 0 as shown in Figures 3.2 and 3.3. At runtime, message passing among tasks that involves interprocessor communication requires DRAFT to perform the mapping from processor id's referenced by the tasks to the actual physical processors. For example, consider once again a DRAFT system containing $N = 5$ CMs and the three applications APP1, APP2, APP3 of Section 3.3. In this scenario, APP1 and APP2 arrive, are loaded, and run. When APP3 arrives, APP2 has already completed and freed up its CMs. Figure 5.4 shows how the entries in the allocation table would appear for these applications.

When an application is loaded, its vector is used to write-enable those RMs paired



APP1

APP1	MP	1	1	0	0	0

APP2

APP2	MP	0	0	1	1	0

APP1 completes

APP3

APP3	FT	1	1	0	0	1

Figure 5.4: Allocation Table Example

with the application's assigned modules.

5.4 Communication with the CMs

The RMs share address and read/write lines, but individual enable lines permit the MC to selectively write or read from each memory. In both write and read operations the data passes through the RLSU. In the write operations the RLSU is configured to accept inputs from the host or the MC and broadcast these inputs onto the N memory data buses. Messages and applications are transferred to the CMs through the RMs. It is worthwhile at this point to look ahead briefly to the role the RLSU plays in message passing. As mentioned earlier, MP tasks address messages to a destination processor using that processor's virtual id as determined from application graph; DRAFT maps this reference to the physical processor: the destination CM. By folding this mapping information into the RLSU at loadtime, the mapping operation occurs directly and without table lookup. This is illustrated in Figure 5.5 where an MP application, APP4, has tasks running on CM 1 and CM 2. Because loading can be done freely into any available processor in DRAFT, virtual processor 1 was assigned to CM 2 and virtual processor 2 was assigned to CM 1. The mapping information is shown folded into the digital network where the virtual-to-physical reference is completed when the RM of the destination processor is write enabled.

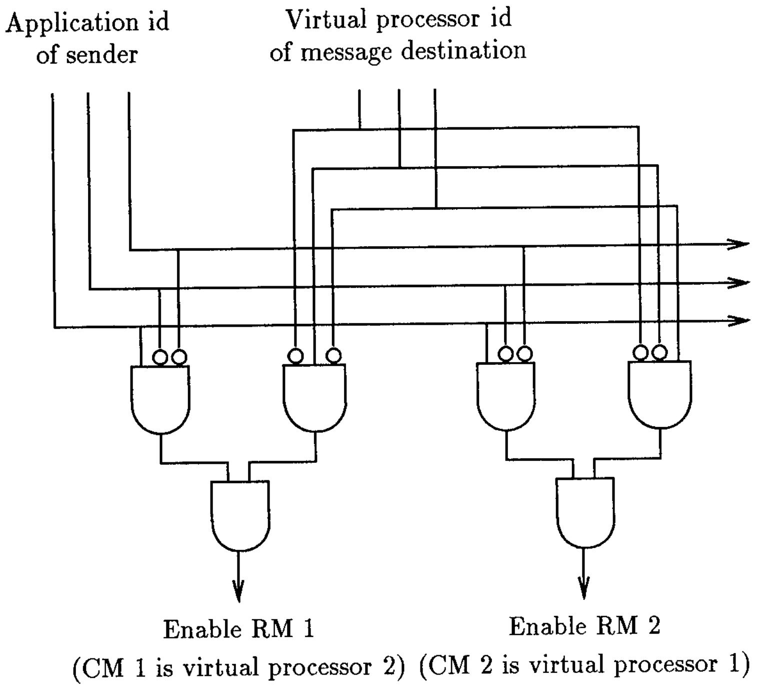


Figure 5.5: Mapping of Virtual-to-Physical Processors

As applications complete, new applications are loaded. As in the case of an example application APP5 that arrives after APP4 has completed. APP5 has virtual processor 3 assigned to CM 1 and virtual processor 4 assigned to CM 2. The RLSU is reconfigured, as shown in Figure 5.6, to reflect this new mapping.

Communication between the MC and the CM is done through messages placed in the RM. This communication system is based on the scheme presented in [33]. A RM's memory locations are all equally accessible by its CM and the MC. The RM is divided

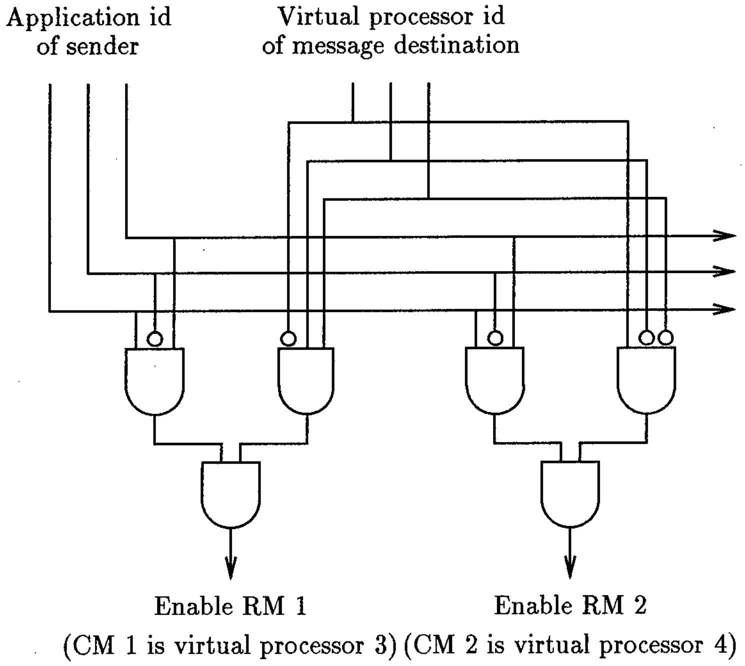

Application id of sender
Virtual processor id of message destination
Enable RM 1
(CM 1 is virtual processor 3)
Enable RM 2
(CM 2 is virtual processor 4)

Figure 5.6: Another Mapping of Virtual-to-Physical Processors

into three regions as shown in Figure 5.7.

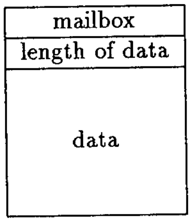


Figure 5.7: Result Memory Format

The first RM region is the mailbox in to which, and out of which, messages are transferred. Messages may be accompanied by data, so the second region is for holding the length of the data stored in the RM. If there are no data with the message, then the contents of this region are set to 0. The third region, which begins immediately after the region holding the data length, contains the data. After a message has been written to the mailbox, a response is required from the receiver before another message is sent. The MC and the CM check the *message_destination_bit* in the mailbox. If this bit is 1, it indicates that the MC has sent a fresh message to the CM in the common memory space. The CM sets this bit to a 0 once it has read the message and responded to the MC. Conversely, when the *message_destination_bit* is 0, then the MC reads the message before it writes to the mailbox.

An application from the host is loaded in the RMs, and, once the load is complete, the MC sends a message to a mailbox in the RM. While awaiting tasks, the CMs poll their mailboxes. Tasks in different CMs can execute concurrently.

5.5 Polling for Task Output

The MC polls the RMs while awaiting task output. The RLSU speeds up the MC polling operation by being configured to recognize the *result_ready* message. The RM mailboxes are read in parallel, and, when the binary code for the *results_ready* message is found in a mailbox, the RLSU raises the appropriate detection signal. Figure 5.8 describes the algorithm used by the MC in polling the RMs for task results. Once the MC knows where the available results are, it can determine what action to take. By comparing the detection signals for *results_ready* with the allocation vectors, the MC decides which task to select and the type of support (MP or FT) that must be provided. In searching the allocation vectors and comparing them against the detection signals, the process of choosing which vector to try next is based on the specific types of applications. Scheduling of service to applications could use any of several well-known algorithms, such as: *fixed priority*; *first-come, first served*; or *round-robin*. For our current implementation, we chose the round-robin scheme.

During the comparison of the allocation vectors with the detection signals, a determination is made as to whether the detection is due to a failed but unassigned CM. If there are multiple CMs for a task and the CMs are not strictly synchronized, the polling algorithm forces a waiting period equal to the predefined maximum skew among redundant tasks before indicating a valid failure detection.

```
Procedure poll_RMs_for_task_output;
  enable = "all RMs";
  if RLSU_configuration $\neq$ "polling" then
    configure_RLSU_for_polling;
  poll_RMs;
  if nonzero(detection) = true then /*presence of RM msg detected*/
    begin /*determine which application responded to polling*/
      index = start; /*select allocation table entry*/
      $i = 1$;
      found = false;
      while $i \leq number\_valid\_entries$ and found = false do
        begin
          allocation_vector = get_vector(allocation_table(index)); /*select a unique vector*/
          enable = (allocation_vector AND detection); /* bitwise AND */
          if nonzero(enable) = true then
            begin
              found = true; /*$\exists$ for allocation_vector and detection a '1' in same bit
                                position*/
              mode = get_mode(allocation_table(index));
              if mode = "MP" and num_ones(enable) > 1 then
                enable = select(enable) /*service one at a time in round robin*/

              /*if FT mode then multiple CMs are used requiring check whether all redundant
                tasks indicate existence of RM msg*/

              if mode = "FT" and enable $\neq$ allocation_vector then
                wait until sync(predefined_max_skew) = true; /*Allow for loosely
                                                              synchronized CMs*/
            end;
          $i = i + 1$;
          index = next_valid_index;
        end;
      if found = false then /*CM not running any task failed and sent erroneous msg*/
        track_faults(in : detection)
      else
        begin /*setup for task output*/
          if mode = "FT" then
            configure_RLSU_for_FT(in : enable); /* for fault tolerance on
                                                   redundant task outputs */
          if mode = "MP" then
            configure_RLSU_for_MP(in : enable); /* for message passing or
                                                   steering application output */
          task_output_handler(in : enable, mode);
        end;
    end;
end poll_RMs_for_task_output.
```

Figure 5.8: MC Polling for Task Output Algorithm

5.5.1 Loose Synchronization of CMs

From characterization of the processors an upper bound on delays between receipt of *results_ready* messages is determined [21]. We call this bound *max_skew*, so that any two fault-free processors participating in a fault-tolerant application each send a *results_ready* message within *max_skew* time units of each other. However, the mere arrival of the *results_ready* does not mean it is valid. The behavior of a faulty processor can be malicious; it can falsely indicate *results_ready* at any time. If all of the non-faulty processors do not indicate *results_ready* within *max_skew* time units from the malicious *result_ready* message, then some will have exceeded the bound, thus giving the appearance that they failed by omitting a *results_ready* message. By causing fault detection to be initiated prematurely, a single malicious failure could effectively cause multiple processor "failures" and completely defeat fault tolerance.

In accordance with the requirements of the application, at most f faulty processors are anticipated and can thus be tolerated. This means that no more than f malicious *results_ready* messages can occur. DRAFT takes action to ensure that fault-detection encompasses all of the fault-free processors so that none are incorrectly declared faulty-by-omission. After the arrival of the initial *results_ready* message, fault detection proceeds for whichever of the subsequent events occurs first: either all participating processors have indicated *results_ready*, or *max_skew* time units have elapsed since the arrival of latest *results_ready* message, or *max_skew* time units have elapsed since the $f+1$ arrival.

Figure 5.9 shows the function that synchronizes n processors based on the arrival of the *results_ready* messages. This function enforces the above stated rule so that a maliciously failed processor cannot invalidate the required fault tolerance.

```
Function sync (in : $max\_skew$) returns boolean;
  $doit$ = false;
  $nfloor2 = n/2$;
  while $doit$ = false do /* loop until timing complete */
    begin
      if $num\_arrivals = n$ then /* all ready */
        /* initiate fault detection */
        $doit$ = true;
      if $doit$ = false then
        begin
          wait for $max\_skew$ until $new\_arrival$ = true;
          if $NOW >= PREV\_TIME + max\_skew$ then   /* max_skew */
          /* initiate fault detection */
            $doit$ = true;
          else
            $PREV\_TIME = NOW$;
          if $num\_arrivals <= nfloor2$ then
            begin
              wait for $max\_skew$ until $new\_arrival$ = true;
              if $NOW >= PREV\_TIME + max\_skew$ then /* no event */
                $doit$ = true;
              else /* an arrival event */
                $PREV\_TIME = NOW$;
            end;
            if $num\_arrivals >= nfloor2 + 1$ and $num\_arrivals < n$ then
              begin
                wait for $max\_skew$ until $n = num\_arrivals$;
                $doit$ = true;
              end;
        end;
    end;
    return($doit$);
end sync;
```

Figure 5.9: Asynchronous Fault Detection Algorithm

5.6 Coordinating Fault Tolerance and Multiprocessing

The focus of our design is on hardware reconfigurability for selective fault tolerance where the MC performs a control function and the RLSU provides direct hardware support for the MP and FT modes.

Application output must be sent to the host. Task results that are application output are routed by the RLSU directly to the host. In the FT mode, the same task will have executed on more than one module so the RLSU is reconfigured to perform a majority or compare function on results from the tasks. The RLSU provides the error-free output and signals when it detects a fault in any module. These error reports are also used by the RLSU to produce a signal for ensuring that a majority of fault-free outputs exists before the result is accepted. The MC also examines the error reports for tracking faulty modules. In the MP mode, task output may have to be transferred to a task in another CM in the form of an intertask message. A task that must send a message to a task in another CM first passes a message to the MC to initiate intertask communication. The MC directs the RLSU to provide the necessary support for the communication to take place. Prior to the actual transfer, however, orderly execution of the processes involved must be ensured.

Modifying the mailbox contents to indicate the presence of a message requires cooperation between the sending and receiving task. In the absence of positive acknowledgment, a message produced and placed in the mailbox by one task may overwrite a fresh message sent from another task. Positive acknowledgment between the communicating tasks prevents messages from being affected in this way; however, the additional communication that is necessary may be unattractive in some applications. An alternate technique [34] is a simpler two-process solution. We apply this solution to ensure that, once intertask communication is initiated by the MC, it will not result in the loss of a message sent between CMs.

The MC and CM processes, P_{mc} and P_{cm}, share the following three variables: $flag[mc]$, $flag[cm]$, and $turn$. Figure 5.10 shows the structure of the algorithm used by P_{mc} to coordinate with P_{cm}. The algorithm for P_{cm} is identical to the algorithm for P_{mc} except that the variables mc and cm are exchanged, and the process P_{cm} checks for new message arrivals before overwriting the mailbox's contents.

```
procedure modify_mailbox (in : $cm$, $message$ ; inout : $mailbox$);
  begin
    $flag[mc] =$ true;
    $turn = cm$;
    wait_until( not($flag[cm]$) or $turn \neq cm$);
    modify_mailbox_contents(in : $message$; out $mailbox$);
    $flag[mc] =$ false;
  end;
end modify_mailbox.
```

Figure 5.10: The Structure of Process P_{mc} to Modify a Mailbox Message

Initially $flag[mc] = flag[cm] = false$, and the value of $turn$ is either mc or cm. The $flag$ and $turn$ variables correspond to memory locations in the RM. Specifically, the mailbox consists of several memory words: three words are used for these variables and the remaining are reserved for mailbox messages. To modify the message portion of the mailbox, process P_{mc} first sets $flag[mc]$ to $true$, and then sets $turn = cm$ so that the CM process can first modify the message if it needs to. If both processes try to modify the message at the same time, an attempt will be made to assign both mc and cm to $turn$. The arbitration logic built into the dual-ported memory allows both writes to occur, but performs them sequentially. The first assignment is then immediately overwritten by the second, so only the second assignment is retained. This value of *turn* decides which of the two processes can modify the message portion of the mailbox.

Process P_{mc} remains in the wait-until loop only if the condition $flag[cm] = true$ and $turn = cm$; however, once P_{cm} executes the modify_mailbox_contents procedure, it will reset $flag[cm]$ to $false$, allowing P_{mc} to exit the loop.

This two-process solution has been proved to be correct and shown to be useful in multiprocessor systems [34]. In DRAFT, this solution is applied to ensure that a newly-arrived message is not accidently destroyed by the intended receiver.

Watchdog timers in the MC prevent a failed CM process from causing the MC to loop indefinitely while waiting for permission to modify the message portion of the mailbox.

The algorithm used by the MC in providing task output service for the MP and FT modes is described in Figure 5.11.

The *enable* vector predetermined by the MC polling algorithm is used to selectively read-enable RMs for task output. Once read, the data are input to the RLSU. From the RLSU, the data can be passed to the host, the MC, or other RMs. For FT tasks, the RLSU performs either a voting or comparison operation. Output from a task in FT mode includes the error detection reports generated by the RLSU. The MC inspects these reports for tracking faulty modules. If no majority exists (NMR), or a miscompare (duplication with comparison) occurs, then the RLSU gives a failure indication to the MC so that erroneous results will not be accepted.

When providing output service to tasks in the MP mode, the RLSU routes application results and task messages. The specific processors involved with intertask

```
Procedure task_output_handler(in : enable, mode);

/*in FT mode must signal condition when number of faulty CMs exceeds fault tolerance
  capability of a task using $n$ ($n \leq N$) CMs*/

  if mode = "FT" then
    on condition( count(detection) $\geq \lceil$ count(enable)/2 $\rceil$)
      send(in : host, "complete task failure, external recovery required");

  read_RMs(in : enable, addr(mailbox); out : msg, detection);
  if mode = "FT" then
    track_faults(detection);
  if msg = $\phi$ then /* failure of a CM running this task falsely initiated this algorithm */
    track_faults(in : detection)
  else
    begin /*valid msg : get destination, length of data, and data*/
      destination = function(msg) /*msg identifies destination (host or virtual processor)*/
      msg = "initiating transmission"; /*outgoing message*/

      /*if MP mode communication between CMs possible — receiving CM determined
        from allocation table*/

      if mode = "MP" and destination $\neq$ host then
        modify_mailbox(in : cm, msg; inout : mailbox) /*process coordination*/
      else
        send(in : destination, msg);
      read_RMs(in : enable, addr(length_of_data); out : length, detection);
      if mode = "FT" then
        track_faults(detection);
      send(in : destination, length);
      address = addr(start_of_data);
      for $i = 1$ to length do
        begin
          read_RMs(in : enable, address; out : data, detection);
          if mode = "FT" then
            track_faults(in : detection);
          send(in : destination, data);
          address = increment(address);
        end;
      send(in : destination, "transmission complete");
    end;
end task_output_handler.
```

Figure 5.11: MC Task Output Algorithm

communication are determined at compile time because DRAFT assigns the applications to the available processors at load time. A task therefore addresses a message to another task using a *virtual* processor id from the application graph. DRAFT must provide rapid mapping of this processor id to the corresponding physical destination processor.

For message passing, the sending task specifies the destination processor as specified by the application graph. Before initiating intertask communication, the MC determines, using the allocation table, the CM that is the destination of the message by translating the processor id of the receiving processor into an enable vector. Once initiated, message transfer then consists of the RLSU receiving the message and forwarding it. Because a message usually consists of several RM data words, transferring a message requires a series of steps where a read of the source RM is followed by a write to the destination RM(s) until the transfer is complete. Locking of an entire RM during message transfer is not necessary. By partitioning the RM into buffers, writes to and reads from distinct buffers limit access contention to the shared mailbox where processes must coordinate so as to ensure that data transfer, as in the form of incoming and outgoing messages, is systematized.

The RLSU, under MC control, enables DRAFT to provide direct hardware support for handling task output in either FT or MP mode.

Chapter 6

Performance Analysis of DRAFT

In this chapter we analyze the performance of DRAFT and the related issues of cost, scalability, extensibility of operable time, and configurability. In all of these analyses, our goal is to ascertain whether the DRAFT architecture is indeed superior to comparable methods. DRAFT is compared with bus-based and point-to-point processor interconnection schemes where the processors themselves are responsible for performing the fault tolerance and multiprocessing support functions. In DRAFT, the RLSU is dedicated to the support of efficient fault-tolerant processing and multiprocessing that are the basic functions of selective fault tolerance, so we also examine the effects of dedicating an entire processor to serve as a substitute for the RLSU.

As prerequisites, the RLSU must implement logical and communication functions for fault masking, fault detection, and interprocessor communication. Avoiding the use of a dedicated unit by offloading these tasks onto the processors themselves has been shown to adversely impact processor throughput in even a small multiprocessor [47]. In the case of the SIFT approach, as much as 60% of the processor's raw throughput was consumed by the software-implementation of the selective fault tolerance functions [26]. It was estimated that the execution of these functions in the MAFT multiprocessor was *two orders of magnitude* too slow for a usable system [27].

Hence, from a performance standpoint the use of a component dedicated to providing selective fault tolerance is absolutely necessary. During this performance analysis, we compare the role of the RLSU as the dedicated unit with that of a very-high speed processor whose *sole* responsibility is to provide the equivalent functionality.

6.1 Performance Model for DRAFT

We now describe how the architecture is modeled and simulated at different levels of abstraction using VHDL. The availability of several VHDL simulators in today's CAE market, offering various price/performance ratios and choices in the target platform [35], has caused use of the language among digital designers to become more widespread. With a design that can be reconfigured on-the-fly during operation, it is vital to consider the behavior during the interim period while new configurations are being downloaded. For this reason, and to guarantee the correctness of sequencing and timing of reconfigurations, full-system simulations are required to assure that the overall systems concept is correct with respect to the requirements.

At this high level, a performance model describes the aspects of the system associated with response time, throughput, and utilization. Because analytical models are intractable for these models (except in special cases), Monte Carlo simulations are necessary to estimate these important performance metrics. Typically, VHDL has not been used for this stage of design. VHDL, which is a powerful system-level simulator [36], supports only deterministic simulation and does not have the necessary capabilities to permit stochastic simulation. However our intent is to create a unified simulation environment that is extensible to greater levels of detail. Our motivations are the following [37]:

- Hierarchical completeness: the behavior of a component must be definable at multiple levels of granularity.
- Separation of concerns: specifications for workload, system-structure, and implementation technology are done separately, that specifications of functional and timing behavior should be separated from specifications of interaction behavior so that both can be individually varied.
- Consistency of interfaces: the interface of a component must be realizable for all levels of granularity of behavior.

We have devised a VHDL model of this architecture that is appropriate for performance analysis. In our technique, a randomized application stream drives the VHDL simulation in such a way as to emulate a stochastic simulator. By post-processing the output data, this model allows us to obtain the desired performance metrics and demonstrate the performance benefits of reconfigurable computing. The application stream generation process is integrated with our computer-aided design (CAD) tools that develop the VHDL-simulatable models. Data obtained from the lower-level simulations determine certain parameters (such as a component's service rate) for the performance model. The details of how these simulations were carried out are discussed in Chapter 7. An analytic performance model of the system has also been created; results from the analytic performance model are used to validate the simulation performance model.

At the highest level of abstraction, the hardware architectural description consists of processing elements, switches, and shared-memory elements. Neither the actual application data nor the operations on these data are described, other than what is required to be known to control the sequence of events. For instance, the amount of output data produced by an application is described simply as a number. Information that describes the characteristics of each application of a stream is contained in a VHDL simulation input file. Characterizing each application for simulation are the following:

- Arrival time: During the simulation, this is the time when the application is scheduled to arrive at the input queue. All times are according to a Poisson process with a chosen arrival rate.

- Mode: An application may be classified as either FT (fault-tolerant) or MP (multiprocessing), so, prior to generating the stream, the desired probability of each mode is declared.

- Number of CMs needed: For a given mode, this is the number of CMs requested by an application. Requests for one, two, three, four, or five CMs, are based on the selected probabilities of occurrence for each. For example, letting random variable R be the number of CMs requested by an application, then $P(R = 1) =$

$P(R = 2) = P(R = 3) = P(R = 4) = P(R = 5) = 0.2$ specifies that requests for one, two, three, four, or five CMs are equiprobable.

- Application size: This is the number of memory words needed to load the application. Application sizes follow a Gaussian distribution with a specified mean and variance.

- Parallel programming model of application: If the mode is MP, then there is a predefined probability that the application was programmed according to either the *data parallel* or *functional parallel* model.

- Message frequency: For an application that is in the MP mode and is programmed using the functional parallel model, this is the number of messages passed between the application's processors. The message passing events are assumed to occur at intervals of equal length during application execution. The frequency of messages per application is given by a Gaussian distribution with a specified mean and variance.

- Amount of output data produced: This is the total number of RM words presented by an application to the RLSU for some type of service. It includes interprocessor messages and application results for the host. Lengths for interprocessor messages are fractions of the total application output based directly on the message frequency. The number of RM words output by an application follows a Gaussian distribution with a specified mean and variance.

- Processing time: Once an application is dispatched and initiated, this is the amount of time that it spends processing. It follows a Gaussian distribution with a specified mean and variance. It does not, however, include the waiting time experienced by the application when resource contention results in delays. By realizing the time and resource dependencies between applications, simulation determines the degree to which the combined behavior of applications produces congestion and queueing.

Concurrent VHDL processes model the parallel behavior of DRAFT's CM/RM pairs and the RLSU with its associated controller and ROM. Figure 6.1 shows the structure of the VHDL model.

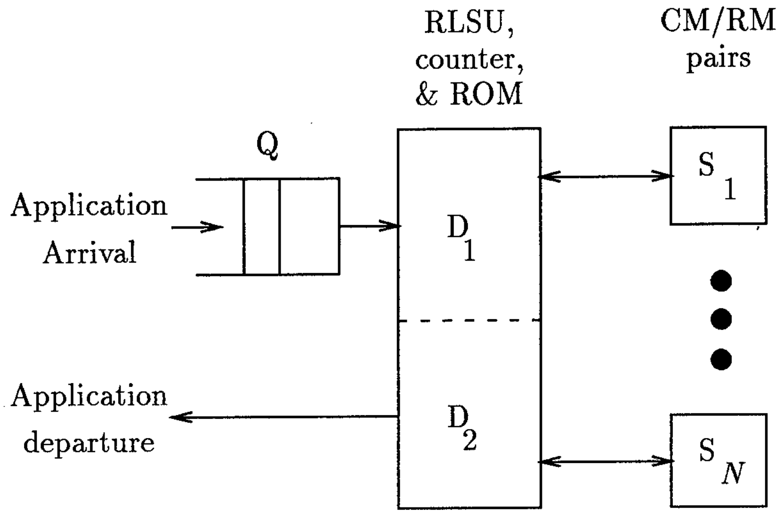


Figure 6.1: VHDL Model

Applications arrive to an input queue (Q) and are dispatched with delay D_1 to a subset of the N CMs (servers S_1 through S_N). Delay D_2 represents the time required by an application for message passing and output service before the application "departs" the system. The input queue is not modeled by a VHDL process because these processes are sensitive to signal events that occur during simulation; unfortunately, the application streams must be created *a priori.* The VHDL simulator itself is used to capture the stochastic behavior of the queue. Comparing the timestamp of the application currently pointed to in the input file with the value of the simulated clock time determines if the entry is treated as an *arrival.* The simulation code is modified so that *TEXTIO* statements in the models write the time and name of relevant events to a history file. Variable-assigned VHDL *wait* statements simulate the reconfiguration and processing delays encountered by applications during queueing and resource contention. The resulting data that are relevant to our performance analysis are tracked and retained.

6.2 Model Verification

6.2.1 Simulation model

A simulation model of the architecture is constructed so that the behavior of the system can be acted out. Only if the architectural model is sensitive to the interaction between its components can the simulation yield realistic results. To make the component models themselves more accurate, their parameters are obtained by back annotation from the more detailed lower-level models in the simulation hierarchy. Stimuli to the architectural model come from a stream of applications; sample application streams can consist of thousands of entries. Because the applications' attributes are determined statistically, we measure the architecture's performance under various loads. DRAFT loads the available applications from the input queue into the CMs, and then initiates execution. Unfolding delays D_1 and D_2 with respect to time transforms the VHDL model of Figure 6.1 into the simulation model of DRAFT shown in Figure 6.2.

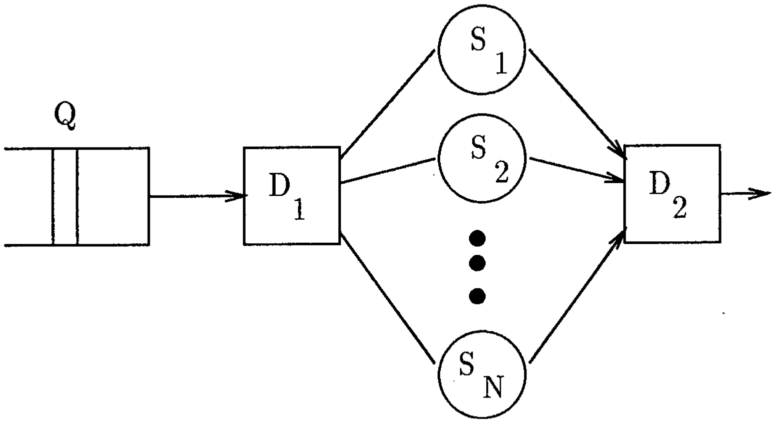


Figure 6.2: Simulation Model

Capturing the causal relationships between components within the architectural model ensures that their combined behaviors will be true to form. It must be verified, however, that these architectural details have been incorporated into the model.

6.2.2 Initial Simulation Model Verification

Once a simulation model and methodology has been established, it is necessary to verify *correctness.* Comparative "walkthroughs" of manually simulated application streams with the results obtained from running the same streams through VHDL simulation revealed no discrepancies. We found that applications completed in the order and at the times anticipated. Having obtained reasonable results, we turned to the problem of verifying that the simulation model is a valid implementation of DRAFT. Prior to developing our own analytic model of DRAFT, we compared our simulation model against a model for which analytic results are already available, as recommended in [38]. By subsetting the simulation model to conform to an established analytic model, we checked for errors during the modeling process. Analytic results for the M/M/1 and M/M/c queueing models are well understood and have been published extensively [39], [40], [41]. Eliminating delays D_1 and D_2 reduces the model of Figure 6.1 to a model consisting of input queue Q and servers S_1 through S_N. Corresponding to D_1 and D_2 are VHDL statements that model the delays caused by RLSU reconfigurations and application support functions. Having the VHDL simulation assign zero time to these activities causes $D_1 = D_2 = 0$.

Application streams were generated such that each application would arrive according to a Poisson process with rate λ and execute according to service rate μ. Tailoring an application stream so that every application requests all servers effectively makes the N servers act like a single server. The result is a simulation model that conforms to an M/M/1 queue. Similarly, when every application of a stream requests only a single server of the N available, then the model conforms to an M/M/c queue where c $= N$.

Several experiments were performed for this stage of the verification. From the analytic model, measures were made (for both model types) using formulas for the average number of applications in the system, L, and the average number in the queue, L_q. For the same values of λ and μ used in the analytic formulas, runs were made of the modified simulation models, and values for L and L_q were calculated. Comparison of the results from the respective analytic and modified simulation models revealed very small differences — less than 5%. The next task was to verify the complete DRAFT simulation model.

6.3 Application Stream Considerations

An important objective of the DRAFT design is to support concurrent applications that are FT or MP. For simplicity, we consider an application to be in only one mode; however, our architecture allows a multiprocessing application to have replicated processors for combined MP *and* FT operation.

The loading of applications was assumed to follow the order of the application stream; out-of-order execution was not exploited. DRAFT maximizes application concurrency by loading as many applications as the number of available CMs can accommodate. The host accepts results as they become available. Regardless of the mode, and assuming all CMs are operational, an application is loaded for concurrent execution with other applications unless by doing so would exceed the total of N CMs. As an example, consider two application streams, A and B, each of length 4, where the *length* of a stream is the number of applications in it. Each application requires a certain number of CMs (ranging from 1 to 4), and assume that $N = 5$ CMs are available. The applications arrive in a particular order, but otherwise simultaneously. In this example, the order of arrival is indicated by a left-to-right sequence of numbers that is the CM requirements for each application stream. For stream A and B we have:

- Application stream A: with CM requirements of 2 3 4 1

- Application stream B: with CM requirements of 1 3 4 2

The total number of CMs required by the two streams is the same, as are the individual numbers of CMs required, but the two streams differ in the *order* of the number of CMs required. Each application is assumed to execute in unit time. When applied to our baseline design of $N = 5$ CMs, the concurrency that DRAFT achieves for these two streams is shown to differ. DRAFT loads the applications of the streams for parallel execution in the following manner (underscores indicate parallel execution):

- Application stream A: with CM requirements are $\underline{2\ 3}\ \underline{4\ 1}$
- Application stream B: with CM requirements are $\underline{1\ 3}\ \underline{4}\ \underline{2}$

Because of parallelism, the applications in streams A and B complete in 2 and 3 time units respectively as shown above. Measuring the CMs' throughput, in number of applications completed in unit time (A/T), for stream A yields 2 A/T and for stream B it is $1\frac{1}{3}$ A/T. This discrepancy shows that the calculation of throughput cannot be based solely on the probability distribution of the number of CM's required for a stream. Furthermore, a non-intuitive result is that the throughput cannot be calculated using the stationary distribution of each number's appearance in the stream. The influence of the immediately preceding numbers is strong — for any requirement other than the full complement of CMs, the preceding numbers dictate the degree of parallelism achievable with the current number. Therefore, the performance obtainable under an anticipated application workload must be analyzed over a long period to allow for order-of-arrival influences to die down and where a steady state has been reached.

6.4 Analytic Model

We now analyze the effective application throughput and latency of DRAFT. The number of applications in a stream is denoted by S_{app}, and the time for DRAFT to complete execution of the the stream is t_{comp}. In order to make the problem analytically tractable, performance measures are based on the rate at which applications complete. Our analysis uses a Markov model for tracking the number of applications completed, and because a Markov model requires single step transitions we make the strong assumption that each application executes in unit time. This assumption can be relaxed for the simulation model where variable completion times are permitted. Nevertheless, identical completion times result in the highest possible contention for output service among concurrent applications. This being the case, our analytic model addresses the most adverse effect that contention has on performance.

The throughput, TP_{DRAFT}, is defined as:

$$TP_{DRAFT} = \frac{S_{app}}{t_{comp}} \tag{6.1}$$

Two time components comprise t_{comp}:

$$t_{comp} = t_{proc} + t_{delay} \tag{6.2}$$

where t_{proc} is the time applications spend receiving service from the CMs, and t_{delay} is the time accrued from MP/FT support. Substituting equation (6.2) into equation (6.1) gives:

$$TP_{DRAFT} = \frac{S_{app}}{t_{proc} + t_{delay}} \tag{6.3}$$

Dividing the numerator and denominator of equation (6.3) by t_{proc} yields:

$$TP_{DRAFT} = \frac{\frac{S_{app}}{t_{proc}}}{1 + \frac{t_{delay}}{t_{proc}}} \tag{6.4}$$

The numerator of equation (6.4) is the average rate at which applications complete CM execution. In Figure 6.3 we show an analytic performance model of DRAFT interactions. The model consists of a FIFO queue, a server, and delays D_1 and D_2 that denote, respectively, the delay incurred by dispatching and then providing output service to applications.

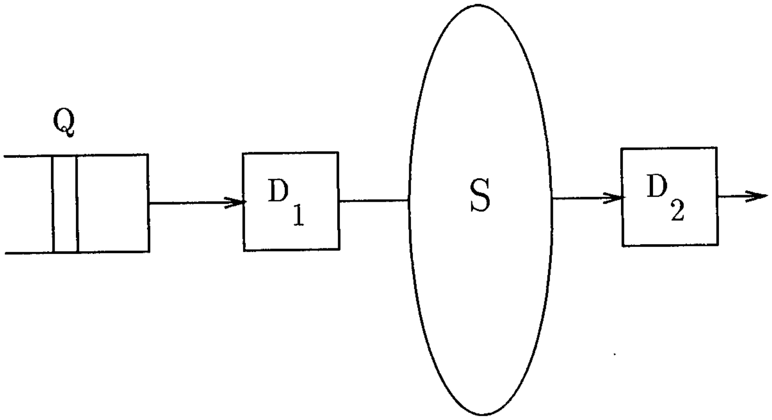


Figure 6.3: Analytic Model

Although this model appears simple, analytic models of queueing systems [39], [40], [41] regularly assume that each service request is satisfied by a private server. Applications in DRAFT, however, actually request a variable number of servers, so, as a result, the server S is a composite of the N individual servers shown in Figure 6.2. The service rate, in number of applications per unit time (A/T), of S in Figure 6.3 is denoted by μ. If μ applications complete in unit time, then the amount of delay these applications experience is $\mu(D_1+D_2)$. Assuming unit time completion for each application (i.e. t_{proc} = 1 time unit), and by making appropriate substitutions, equation 6.4 becomes:

$$TP_{DRAFT} = \frac{\mu}{1+\mu(D_1+D_2)} \tag{6.5}$$

The latency of DRAFT, denoted by LT_{DRAFT}, is given by:

$$LT_{DRAFT} = 1+\mu(D_1+D_2) \tag{6.6}$$

Both D_1 and D_2 are pure delay. Low-level design details of logic components supply the deterministic values for D_1 and D_2. Application statistics, such as the mean number of message passing events, dictate the frequency of particular logic component functions. Characterizing S, however, cannot be done using a static probability distribution. Next we apply hierarchical modeling to determine μ.

6.4.1 Hierarchical Analytic Model

The general rule for applying queueing theory stipulates that a customer waiting in the queue will request a private server. Because applications (customers) request a variable number of individual CMs (servers), analysis of the composite server S is not easily done by a direct application of queueing theory. Instead, we apply a Markov process where a discrete-parameter Markov chain captures the behavior of S. Customers awaiting service are dispatched to the available servers, and then service commences. Service is granted for as many customers as permitted by the number of individual servers. Two types of states make up the chain: the *Have Total (HT)* and *Execute-Pending (EP)* states. A HT state tracks the number of servers that have been assigned to customers. Eventually, as many customers as possible have servers assigned to them. At this point service commences, and the customer at the head of the queue has service pending. We call this situation an EP state. The number of servers requested by the customer at the head of the queue who was temporarily denied service is used to identify the EP state. Figure 6.4 shows a Markov chain for the case where S is comprised of two servers, and state transition probabilities P_1 and P_2 correspond to the probability that 1 or 2 servers, respectively, are requested.

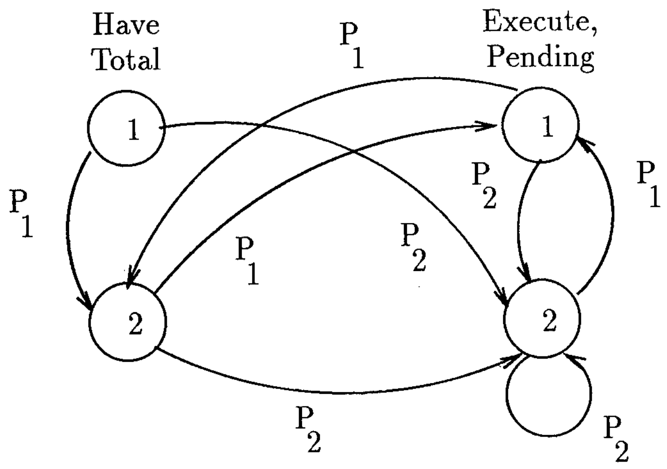


Figure 6.4: Markov Chain for the Two-Server Case

Figure 6.5 shows the transition probability matrix, T, for the general case of N individual servers. Next, we calculate the steady state for the Markov chain. In matrix notation:

$$\mathbf{v} = \mathbf{v}T \tag{6.7}$$

$$
\begin{array}{cc|cccccccccc|}
 & & \multicolumn{5}{c}{\overbrace{\hspace{12em}}^{\text{Have Total}}} & \multicolumn{5}{c}{\overbrace{\hspace{12em}}^{\text{Execute-Pending}}} \\
 & & 1 & 2 & \cdots & N\text{-}1 & N & 1 & 2 & \cdots & N\text{-}1 & N \\
\hline
 & 1 & 0 & P_1 & \cdots & P_{N-2} & P_{N-1} & 0 & 0 & \cdots & 0 & P_N \\
 & 2 & 0 & 0 & \cdots & P_{N-3} & P_{N-2} & 0 & 0 & \cdots & P_{N-1} & P_N \\
\text{Have Total} & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\
 & N\text{-}1 & 0 & 0 & \cdots & 0 & P_1 & 0 & P_2 & \cdots & P_{N-1} & P_N \\
 & N & 0 & 0 & \cdots & 0 & 0 & P_1 & P_2 & \cdots & P_{N-1} & P_N \\
 & 1 & 0 & P_1 & \cdots & P_{N-2} & P_{N-1} & 0 & 0 & \cdots & 0 & P_N \\
 & 2 & 0 & 0 & \cdots & P_{N-3} & P_{N-2} & 0 & 0 & \cdots & P_{N-1} & P_N \\
\text{Execute Pending} & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots & \vdots & \vdots & \cdots & \vdots & \vdots \\
 & N\text{-}1 & 0 & 0 & \cdots & 0 & P_1 & 0 & P_2 & \cdots & P_{N-1} & P_N \\
 & N & 0 & 0 & \cdots & 0 & 0 & P_1 & P_2 & \cdots & P_{N-1} & P_N \\
\hline
\end{array}
$$

Figure 6.5: Transition Probability Matrix - *T*

we find $\mathbf{v}$ — the left eigenvector of T — where

$$\mathbf{v} = [v_{HT_1}, \ldots, v_{HT_N}, v_{EP_1}, \ldots, v_{EP_N}] \tag{6.8}$$

The Markov chain is aperiodic, so for the n-step transitions

$$v_j = \lim_{n \to \infty} P_j(n) \text{ exists, where } j \in \{HT_1, \ldots, HT_N, EP_1, \ldots, EP_N\}$$

The probability v_j is interpreted as the long-run proportion of time the Markov chain spends in state j. This is also the *frequency* of visiting state j, and

$$\frac{1}{frequency} = period$$

is the average number of states visited to reach an *Execute-Pending* state.

The frequency of being in an EP state is $\sum_{i=1}^{N} v_{EP_i}$. Thus, the expected throughput of applications with unit-time length service times, for a given application stream, is:

$$\mu = \frac{1}{\sum_{i=1}^{N} v_{EP_i}} \tag{6.9}$$

Values for μ depend on an application stream's demand for servers. If every application requires N servers, then for this stream $\mu = 1$. At the other extreme, if every application requests only one server, then $\mu = N$. Therefore, we first consider classifying application streams according to the unique number of servers requested by each application in the stream.

The possible number of servers requested by each application is an element of the following set:

$$C = \{1, 2, 3, \ldots, N-1, N\}$$

The set of all *unique* requests for *numbers* of servers made during execution of an application stream is therefore a subset of C. Among all applications in a stream however, some number of servers will not be requested. Taking this into account, the types of application streams to consider are based on the number of servers requested per each application. The family of all subsets of any C is called the *power set* of C, and is denoted by $\mathcal{P}(C)$. $\mathcal{P}(C)$ has 2^N elements. The empty set is an element of $\mathcal{P}(C)$, and because an application must request at least one server,

$$M = \mathcal{P}(C) - \{\emptyset\}$$

is the set of subsets of C representing the different number of servers that can be requested by each application during the course of an application stream. The number of elements in M is obviously $2^N - 1$.

Now, we consider two versions of our analytic model — one for each mode of DRAFT operation. Previously, we noted that support for uniprocessing is covered in DRAFT by simply declaring that a single processor is needed for an MP application. The number of servers requested by an MP application can therefore be any element of C, and the set of all such requests made in some stream is an element of M. The need to tolerate hardware faults while in FT mode necessitates that each application request at least two servers. Applications in this mode can therefore request a number of servers only from the set $C - \{1\} = \{2, 3 \ldots, N\}$.

The set M represents server requests in MP mode, so the set $F = \mathcal{P}(C - \{1\}) - \{\emptyset\}$ contains all sets of unique requests for servers found in any stream of FT applications. The number of elements in F is $2^{N-1} - 1$.

Next, we carry out the analysis for the particular DRAFT that we designed and simulated.

The DRAFT that we have designed has $N = 5$ CMs (servers). For FT mode, there are $2^{5-1} - 1 = 15$ types of application streams that drive server demands. In MP mode, there are $2^5 - 1 = 31$ types of application streams considered. For each element sets of M and F we generate the transition probability matrix. For the transition probabilities:

$$P_i = P_j, \text{ for } i, j = 1, 2, \ldots, N \text{ and } P_i \neq 0$$

That is, in the set of number-of-servers-requested we assume that the elements are equiprobable.

We now analyze the throughput and latency based on our hierarchical queueing model, and validate the results by simulation. By considering all possible requests for servers that are placed on DRAFT, we characterize μ. For each mode, all possible stream types are considered. Sets M and F are constructed. The left eigenvectors are then calculated, and values for μ are obtained. Values for D_1 and D_2 are assigned based on the frequency that their composite functions are called in direct relation to the applications stream's statistics. Equations 6.5 and 6.6 give values for TP_{DRAFT} and LT_{DRAFT} respectively.

The elements of M and F are themselves sets. We index the elements of M and F, according to non-decreasing values of TP_{DRAFT} obtained when these elements are applied to our analytic model. Figure 6.6 shows the indexing of M's 31 elements. The 15 elements of F are indexed as shown in Figure 6.7.

Index	Element of M
1	{5}
2	{4, 5}
3	{3, 4, 5}
4	{3, 5}
5	{4}
6	{3, 4}
7	{3}
8	{2, 4, 5}
9	{2, 3, 4, 5}
10	{2, 5}
11	{2, 4}
12	{1, 3, 4, 5}
13	{2, 3, 5}
14	{2, 3, 4}
15	{1, 4, 5}
16	{1, 5}
17	{1, 3, 5}
18	{1, 2, 3, 4, 5}
19	{1, 2, 4, 5}
20	{1, 3, 4}
21	{1, 2, 3, 5}
22	{1, 2, 5}
23	{1, 2, 3, 4}
24	{2, 3}
25	{1, 2, 4}
26	{1, 4}
27	{1, 3}
28	{2}
29	{1, 2, 3}
30	{1, 2}
31	{1}

Figure 6.6: Indexing Elements of M

For both analytical analysis and simulation, we use parameters that are realizable. We assume that applications with unit completion times arrive at the input Q according to a Poisson process with identical arrival rate λ. For a stable system, the condition $\lambda < \mu$ must hold, and we use the same values of λ in the analytical model and in the stream

Index	Element of F
1	{5}
2	{4, 5}
3	{3, 4, 5}
4	{3, 5}
5	{4}
6	{3, 4}
7	{3}
8	{2, 4, 5}
9	{2, 3, 4, 5}
10	{2, 5}
11	{2, 4}
12	{2, 3, 5}
13	{2, 3, 4}
14	{2, 3}
15	{2}

Figure 6.7: Indexing Elements of F

generation process that will drive the simulation. Application streams consisting of 10,000 entries were created. Postprocessing of simulation results provided close estimates of TP_{DRAFT} and LT_{DRAFT}.

Figure 6.8 shows the application throughput for MP mode obtained from both analysis and simulation with respect to the indices of set M.

For the first seven indices, every element of M contains nothing smaller than 3. For these indices no application concurrency is possible and throughput cannot exceed 1 A/T. At higher indices, requests for 3 or more CMs are combined with requests for 2 or fewer so that application concurrency is possible. Maximum application concurrency, and thus throughput, is achieved when requests are for only one CM per application.

Figure 6.9 shows the application latency for MP mode obtained from analysis and simulation again with respect to the indices of set M. This shows the delay encountered by applications once they leave the input queue, as a function of their index as given

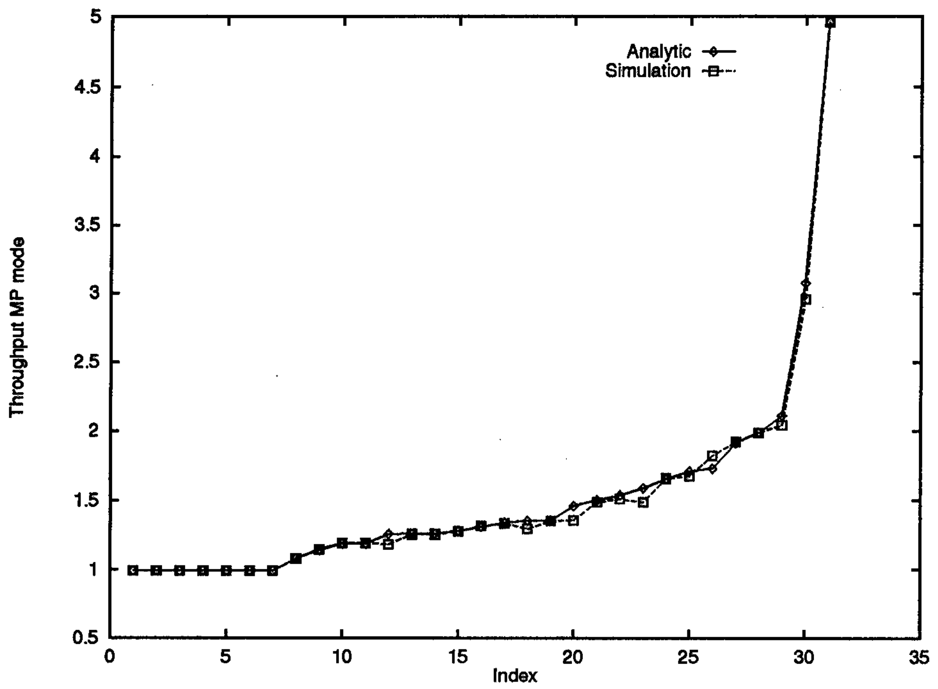


Figure 6.8: MP Mode: Throughput vs. Index

in Figure 6.6. In the data parallel model of the MP mode, tasks are identical and are dispatched in parallel. Although this is not the case for the functional parallel model, the size of the tasks did not produce significant differences in the dispatch time. For both types of MP applications, the processing time and the amount of data output were the same so that latencies are not markedly different across the indices.

Figures 6.10 and 6.11 show performance of FT applications in terms of throughput and latency respectively. Both of these measures are based on the indices of set F.

For reasons similar to those that apply to the MP mode, throughput cannot exceed 1 AT when no fewer than three CMs are requested in a stream. When requests of this magnitude are offset by requests for two CMs, as in the higher indices, then application concurrency boosts throughput.

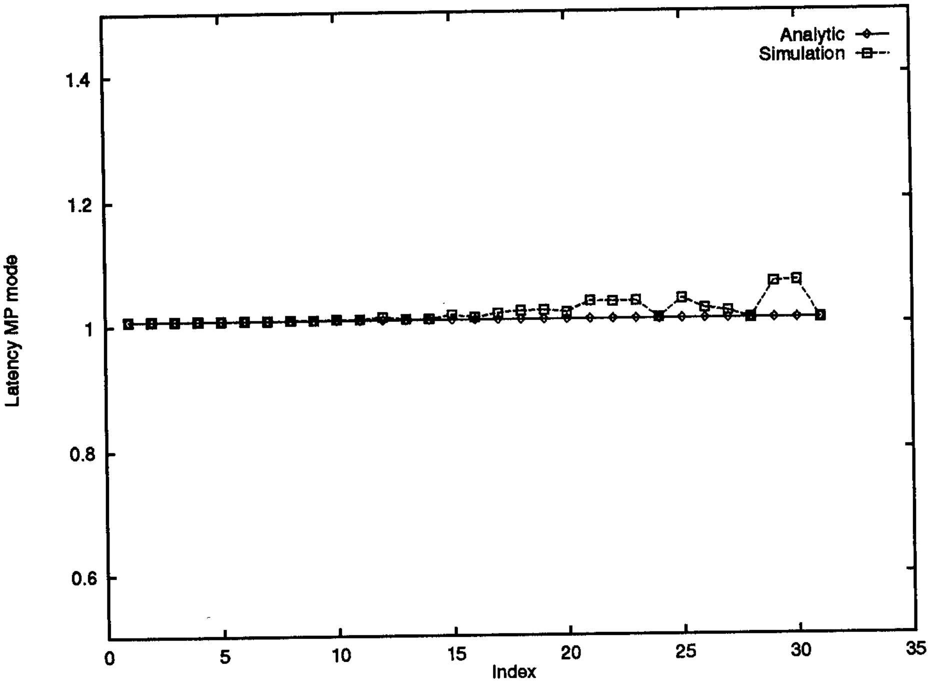


Figure 6.9: MP Mode: Latency vs. Index

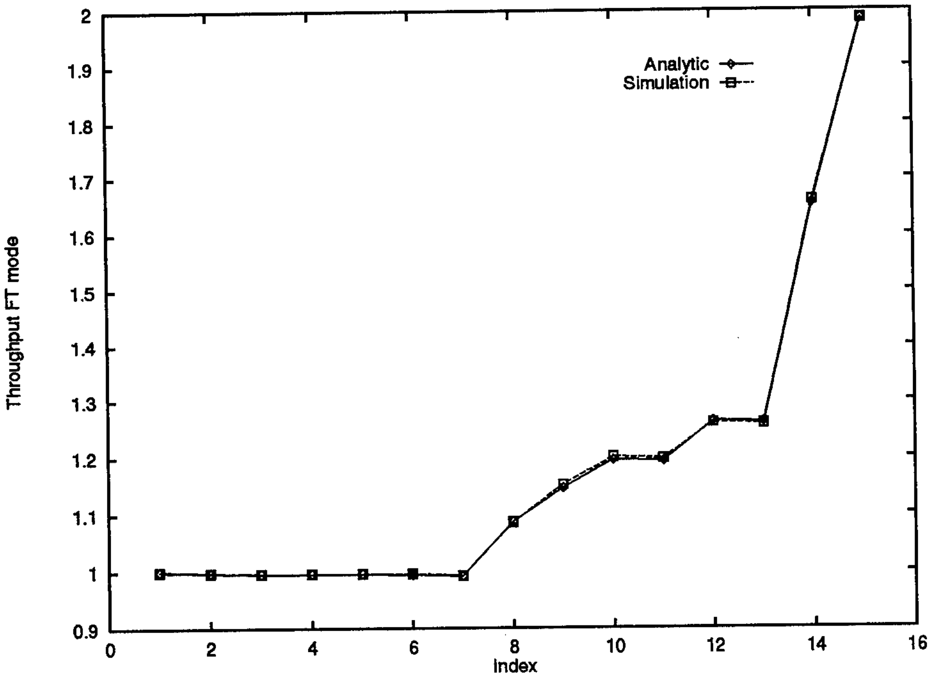


Figure 6.10: FT Mode: Throughput vs. Index

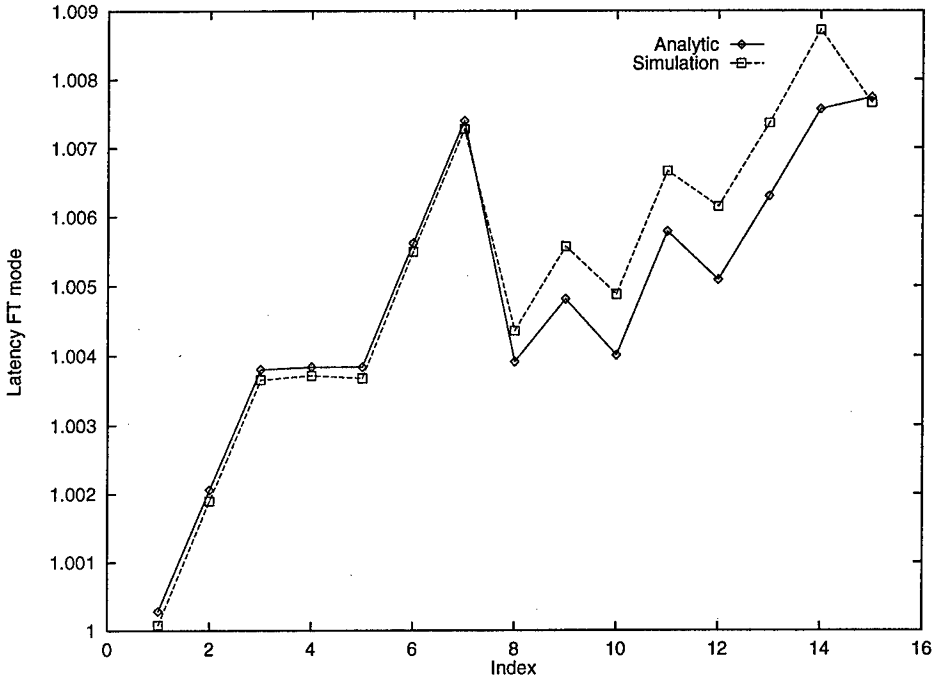


Figure 6.11: FT Mode: Latency vs. Index

One of the factors that influences latency is the reconfiguration time for the RLSU. From the RLSU layout described in the next chapter, we have determined the times required to partially reconfigure the baseline 5MR function to another FT function of fewer inputs. Upon completion of the function, the reconfiguration strategy calls for returning to the function of five inputs. The longest reconfiguration time, based on our implementation, is when the five input function is transformed to the three input function and then back again.

Our approach is to provide the best performance for streams with all different CM requirements. Without prior knowledge of an application stream's makeup, the reconfiguration strategy cannot be tailored to optimally match the requirements of the stream. Index seven of set F involves applications requiring only triplication of CMs. Following the general reconfiguration strategy, with each application, the RLSU is changed from a 5MR function to a 3MR function and back again. Propagation delays through the RLSU structure for functions of different numbers of inputs vary by only tens of nanoseconds so the impact of this factor on latency is negligible. All else being equal, the overiding cause for differences in latency is the differences in reconfiguration time.

It is obvious from the data presented in Figures 6.8 – 6.11 that the simulation results track extremely closely with those of the analytic models for both the MP and FT modes. Next, we compare the performance of DRAFT against that which would be achievable if ultra-high speed general-purpose processors were used instead to implement the functions provided by DRAFT.

6.5 Comparative Performance Results

Previously, we have justified the DRAFT architecture as being the best suited for the dual role of multiprocessing and fault tolerance. Multiprocessors can configure their processors for fault-tolerant operation and distribute the vote among them. In shared-bus multiprocessors the serial nature of the bus impedes the voting process. Fully connecting the processors is an alternative solution, but multiple connections complicate each processor's interface. In either case, however, when the processors themselves provide the comparison and error detection operations for fault tolerance, then fault tolerance internal to the processors must be added, or assumptions that severely restrict the types of faults tolerated must be made. Otherwise, a faulty processor that is responsible for producing the system output is a single point of failure for the system, and its failure rate will dominate the overall failure rate of the system. Therefore, the unit that is the output stage of a fault-tolerant system must offer high performance and high reliability. In this section we show that the DRAFT design offers the highest performance in meeting its goal.

Application streams each consisting of 10,000 applications were simulated. For applications in FT mode, requirements for 2, 3, 4, or 5 CMs (for any application) were specified as being equiprobable. Interprocessor communication implies multiple processors, so, for MP applications programmed according to the functional parallel model (hereafter referred to as being in the MPFP mode) at least two CMs are needed. Requirements of 2, 3, 4, or 5 CMs for any application in the MPFP mode were equiprobable. Applications conforming to the data parallel model (hereafter referred to as being in MPDP mode) each had equal probability of requiring 1, 2, 3, 4, or 5 CMs. Streams were created to allow applications to continuously arrive at DRAFT such that an application was always waiting to be dispatched, (i.e. the input queue was always full) and the mean completion time for an application, once dispatched, was assumed to be 10 ms.

For the purpose of comparison, we consider here the effects of an alternative design: replacing the DRAFT module with a processor that is dedicated to performing the same fault tolerance and multiprocessing support as the RLSU. Equivalent software versions of the RLSU functions were written in an assembly language [42] and executed to verify their correctness. This was done to provide a conservative comparison that gave every advantage to the alternative design. Based on the instruction counts, the corresponding completion times for processors with specified MIPS (Millions of Instructions Per

Second) ratings are also shown in the following figures.

Figure 6.12 shows the throughput of applications in MPFP mode as a function of the *amount of data* produced by each application. Message passing is part of MPFP operation, and in the situation described in this graph, each application passes eight messages among its CMs.

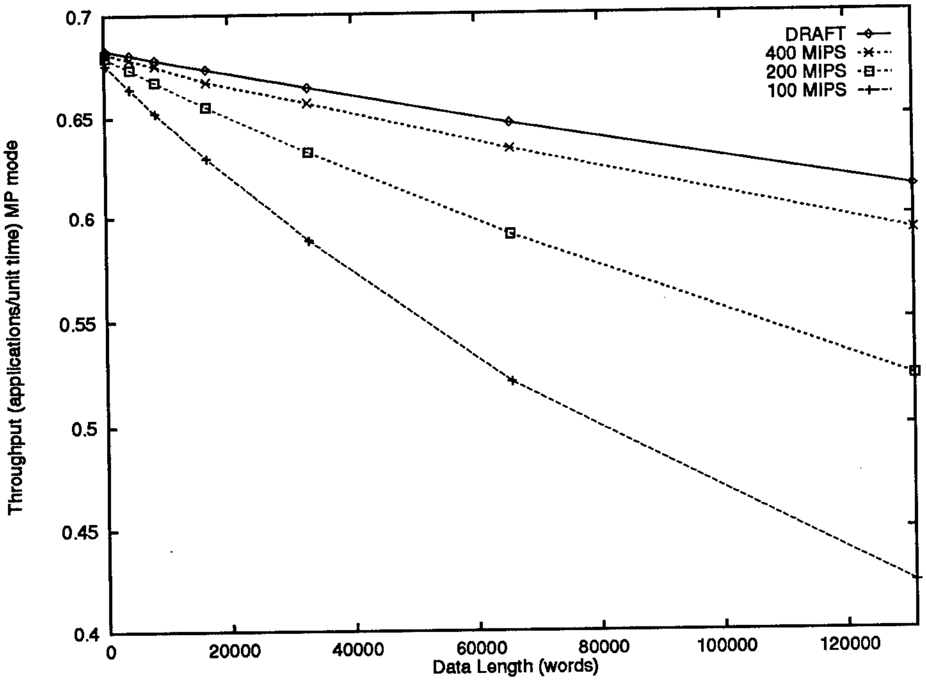


Figure 6.12: MPFP Mode: Throughput vs. Data Stream Length

Figure 6.13 shows the impact of *message frequency* on throughput. The data length for each application was fixed at 16,384 words. This graph shows that, as message frequency is increased, throughput quickly drops. More importantly, throughput in this case is unaffected by the processors' MIPS ratings as these curves are indistinguishable. Synchronizing the CMs to coordinate their message passing activity consumes a considerable amount of throughput. In spite of dedicating a very-high speed processor or even the RLSU to facilitate the exchange of information, there is no marked improvement. In light of this finding, it is clear that as a rule communication between CMs should be made using a few long messages rather than with many short ones.

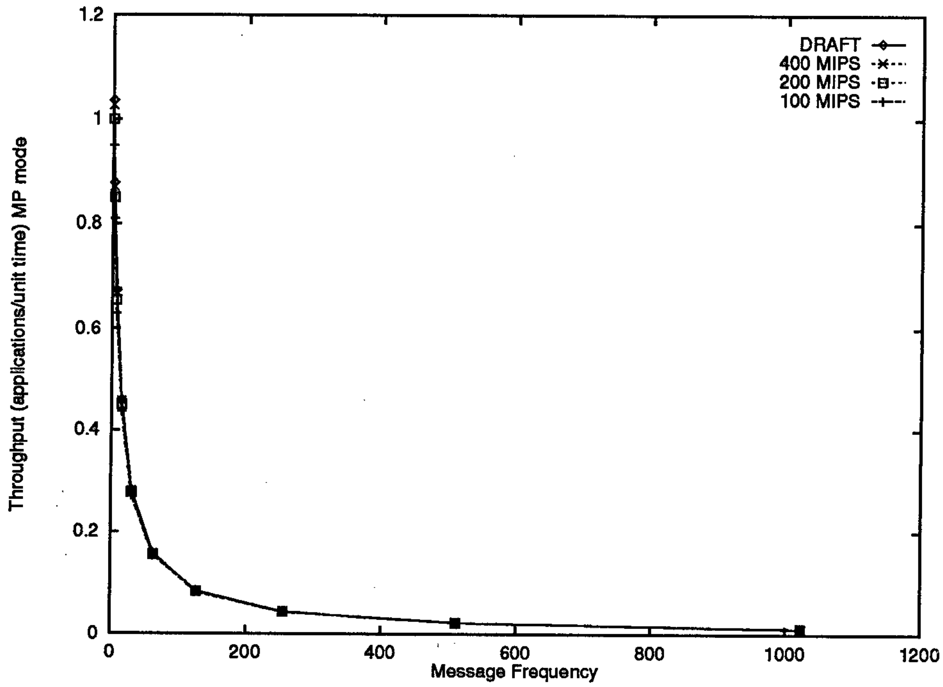


Figure 6.13: MPFP Mode: Throughput vs. Message Frequency

Figure 6.14 shows the corresponding application latency for the MPFP mode. We again assumed a constant count of eight messages per application.

Figure 6.15 shows the throughput of applications in the MPDP mode versus the data length, while Figure 6.16 shows the latency.

Applications in the FT mode yielded the results shown in Figures 6.17 and 6.18. The dramatic improvement provided by the FPGA-base RLSU is clearly shown here.

Finally, the throughput and latency for an equal mixture of applications in the MPFP (with the message count per application set at 8), MPDP, and FT mode are

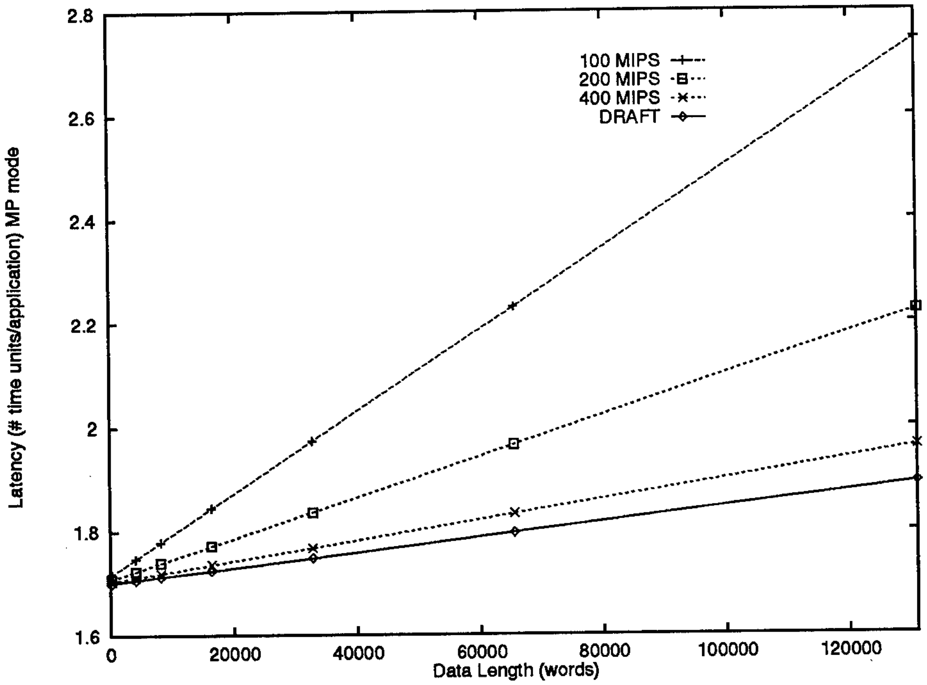


Figure 6.14: MPFP Mode: Latency vs. Message Frequency

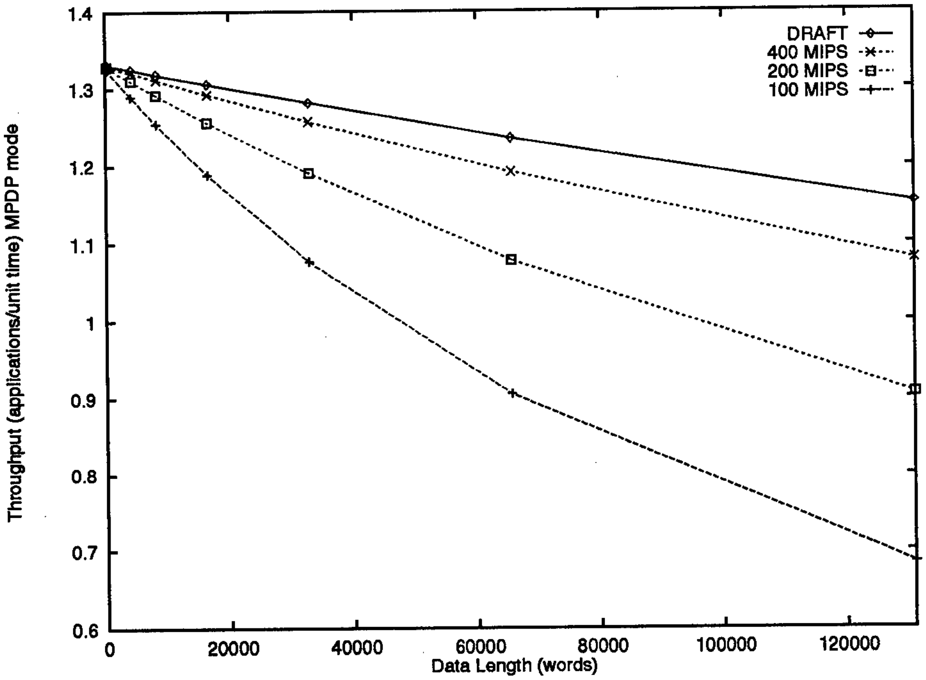


Figure 6.15: MPDP Mode: Throughput vs. Data Stream Length

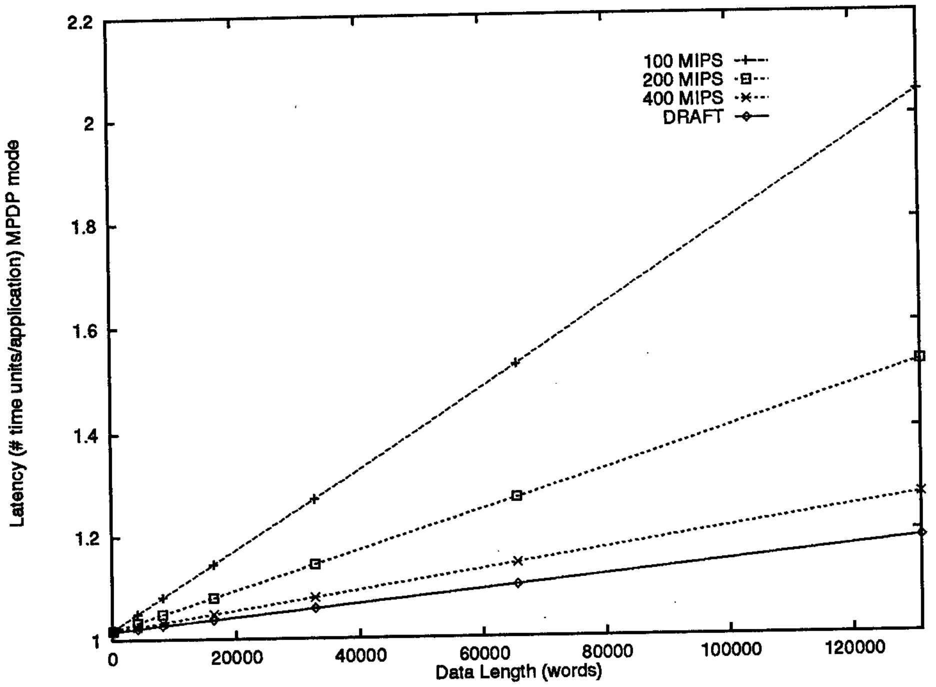


Figure 6.16: MPDP Mode: Latency vs. Data Stream Length

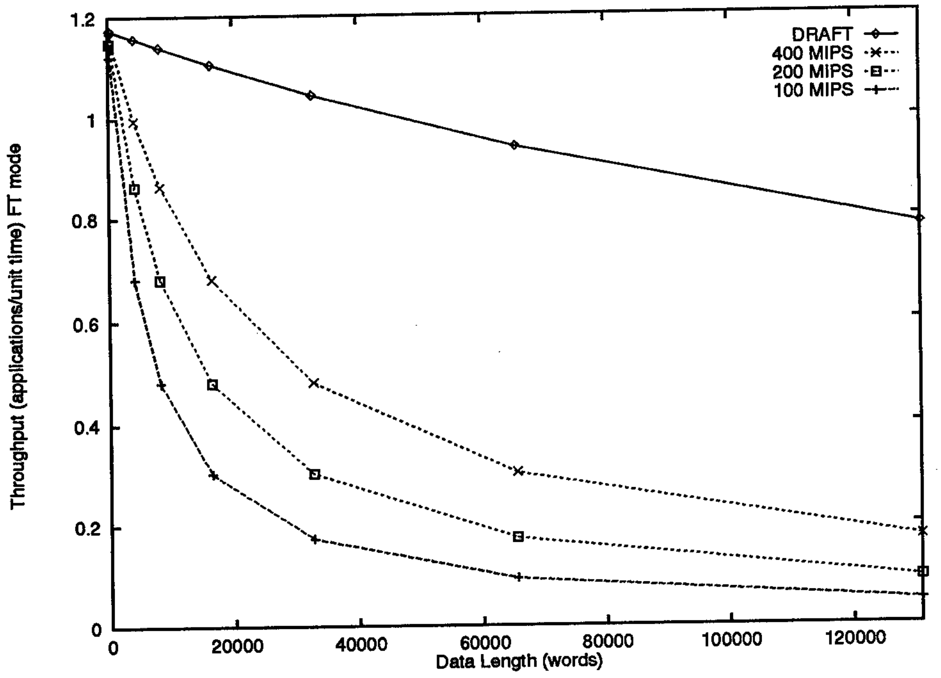


Figure 6.17: FT Mode: Throughput vs. Data Stream Length

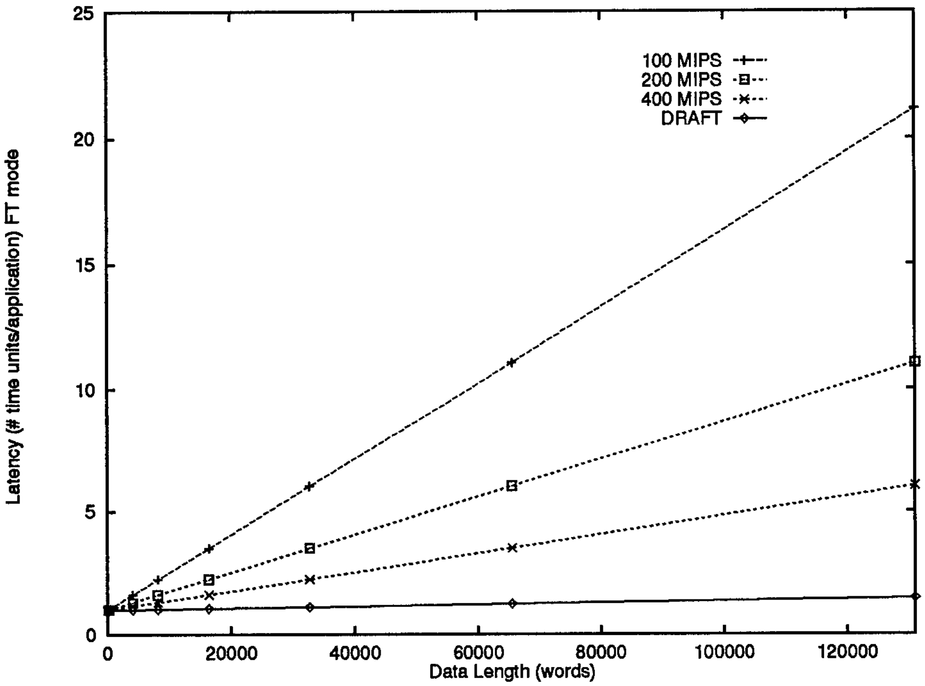


Figure 6.18: FT Mode: Latency vs. Data Stream Length

shown in Figures 6.19 and 6.20 respectively.

These figures demonstrate the potential performance gains obtainable with the DRAFT architecture. It is important to note that these speedups are not just for the overhead functions required to *manage* the FT and MP operations, but are in fact the speedups of the applications themselves.

As the sheer computing power of processors continues to grow, even greater data lengths will be producible using equal or less processing time. The trends presented in these figures indicate that the throughput and latency advantages of DRAFT become even more dramatic with increasing data lengths. Thus the DRAFT design is well situated to meet the future needs of high-performance multiprocessing and fault tolerance.

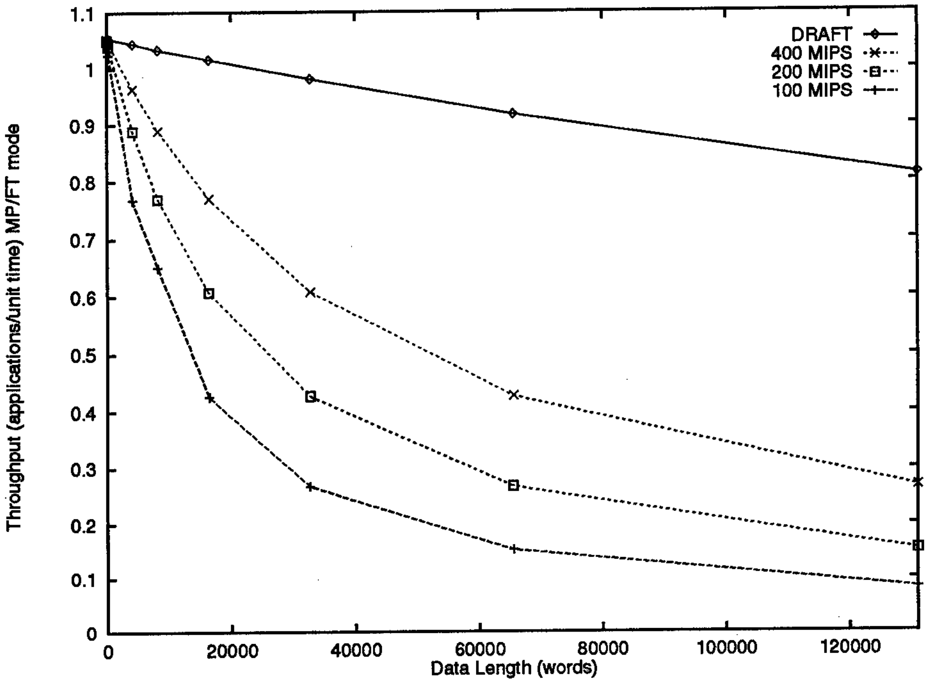


Figure 6.19: MP/FT Mode: Throughput vs. Data Stream Length

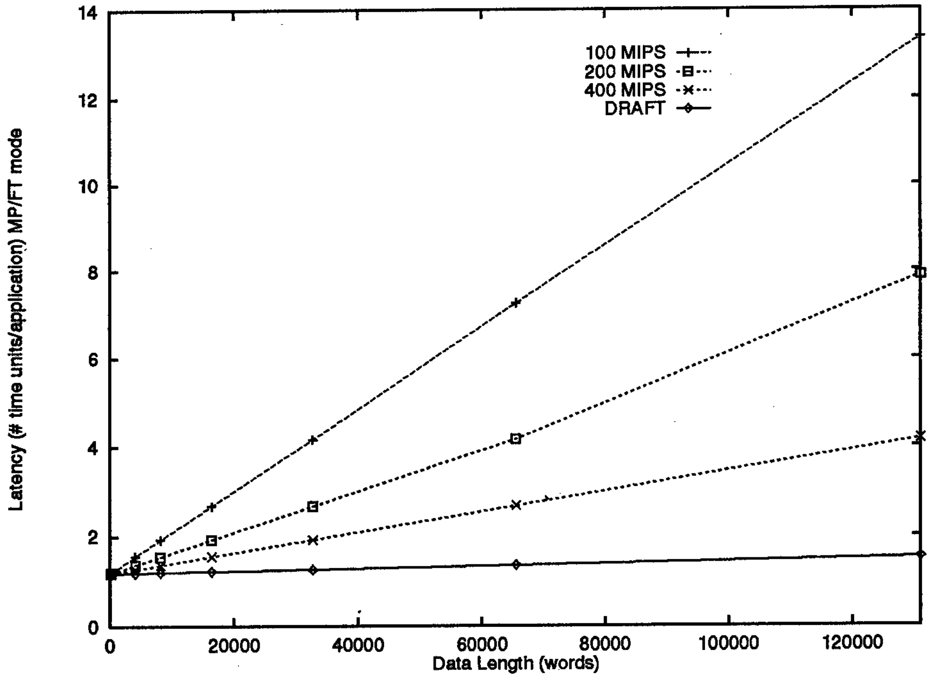


Figure 6.20: MP/FT Mode: Latency vs. Data Stream Length

The RLSU's performance overtakes that of any reasonable implementation based on a dedicated general-purpose processor. It is important to note that a dedicated processor with the performance assumed here would not be a simple, low-cost device. It would require, at the least, a relatively large number of packages to provide the core functions (including memory management), plus substantial local memory. In contrast, there is a practical, highly-reliable, and relatively low-cost enabling technology for the RLSU. Applying this technology to the RLSU, however, is the subject of Chapter 7.

In the next section the cost of DRAFT and comparable architectures is analyzed. The analysis shows that, even when a dedicated processor is altogether avoided by using the processors themselves to perform the selective fault tolerance functions, DRAFT still achieves much better performance at lower cost.

6.6 Cost Comparisions

We compare the cost of DRAFT with two types of multiprocessor interconnection schemes: (1) a *bus-based* multiprocessor where the communication medium is shared among all of the processors; (2) where processors are connected by *point-to-point communication channels.* Figure 6.21 shows the general topologies for each of these two schemes.

Multiprocessing and fault tolerance can be supported by either of these schemes, but because fault tolerance entails physical replication of hardware for redundancy it is the main cost driver. Algorithms achieve fault tolerance by resolving the redundant processor outputs to produce an error-free output. In the schemes shown in Figure 6.21,

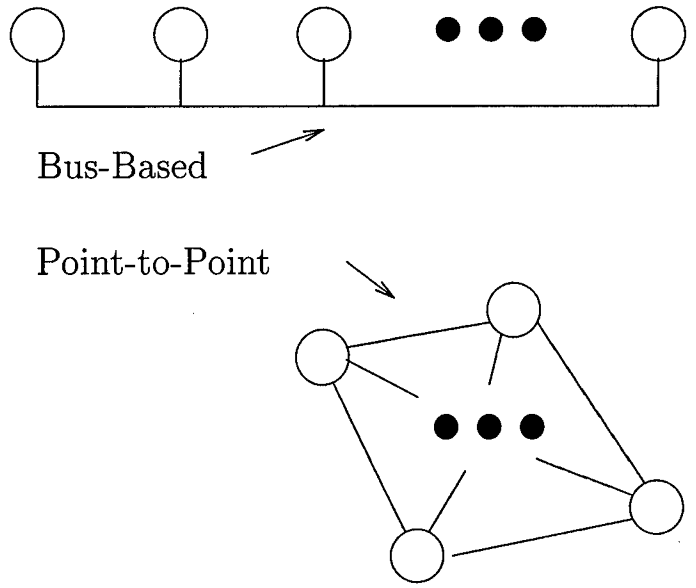


Figure 6.21: General Topologies

the algorithms are implemented by software running on the processors.

A necessary feature of DRAFT is the use of dual-ported memories for the RMs. Use of extra memory for fault tolerance is not, however, exclusive to DRAFT. In software voting [3], each processor writes results to its own dual-ported memory that are then read by the other processors. In this way, all processors can vote independently on the results. We consider the memory costs for storing results to be equal for DRAFT and each of the two multiprocessing schemes.

Bus-based connections pose several drawbacks to processors running fault-tolerant applications. For example, the occurrence of a bus fault has high probability of disabling the entire system, or a faulty processor's behavior might result in it acquiring the bus and never releasing it. Bus redundancy is only part of the solution; each processor must possess the capability to "fail-safe" by detecting its own faults and then isolating itself from the bus. Otherwise, a failed processor may disable the entire redundant bus so that all communication among the working processors stops. Additional hardware is attached to each processor for the purpose of making the processor fail-safe. The hardware cost for fail-safe processing, particularly with off-the-shelf components or systems, is likely to be at least twice the normal cost. This is because duplication and lock-step operation with output comparison is the most straightforward and perhaps the only available method for on-line fault detection. Additional logic for bus isolation further drives up the cost. In DRAFT, the fault confinement area consists of the CMs, the RMs, and the data buses from the RMs to the RLSU. Confining the erroneous data caused by these faults to these components is done without changes to the CM hardware. Intervention by the RLSU prevents faulty processors from influencing correct DRAFT operation. Any bus-based scheme where processors provide the fault tolerance operations themselves, and the processors are directly interconnected, suffers from the possibility of a malicious processor fault corrupting the entire system. The cost of making such a system fault-tolerant through fail-safe processor operation is double that of DRAFT in terms of the number of *processors.* Without sacrificing fault tolerance, DRAFT halves the number of processors required while only slight overhead is accrued. The overhead consists of the dedicated unit composed of the RLSU, RLSU controller, and ROM. We relate component complexity (i.e., gate count) directly to cost, and in Chapter 7, we show that the dedicated unit has less complexity than the type of high-performance processor that would need to be employed as a CM.

A component's complexity is also directly related to its unreliability. The Mil-Hdbk-217E reliability model [71] applied in Chapter 7 measures the complexity of a digital circuit using its gate, bit, package, and pin counts to predict the circuit's failure rate, λ.

Interconnection schemes that are not bus-based, but instead use point-to-point connections between processors, are not vulnerable to a single processor fault completely disabling all communications. Individual links between processors can be made redundant in order to tolerate a link failure. Because there is no single continuous link joining all of the processors, a failed processor that disables its links to other processors does not necessarily prevent communication among the remaining working processors in the system. To achieve this goal, all that is required is to provide adequate connectivity in the initial network topology so that communication can continue in spite of breaks in the structure that correspond to failed processors or their links. Failures can occur at any time, so entrusting the fault tolerance to a single processor increases the vulnerability of the system. As in the bus-based connection scheme, assigning majority voting or comparison operations to a single unit that has the same complexity, or even the same *order* of complexity, of one of the redundant processors has the effect of limiting the system reliability to the reliability of one of its composite processors. Redundancy in this case is detrimental; the simplex approach offers better reliability and at lower cost. To make the fault tolerance operations reliable, they are carried out redundantly on multiple processors. However, when processors must cooperate among themselves, as in performing a majority vote on their outputs, another type of failure can occur that is often overlooked: namely, sending conflicting information to different processors. This is caused by the so-called Byzantine fault [6].

The theoretical requirements necessary to guarantee correct system behavior in this situation can be summarized as follows. In order to tolerate f Byzantine faults, it is necessary to have $3f + 1$ independent participating processors in the Byzantine-fault-tolerant scheme, where the processors are connected by $2f + 1$ disjoint communication paths, and go through $f + 1$ rounds of information exchange to arrive at exact consensus. Byzantine faults are the most malicious kind of processor faults that can be considered, and therefore are the most difficult to tolerate.

In a cost analysis, we consider the number of processors required in the DRAFT approach. We compare this to other approaches where a dedicated processor is not used, which we refer to as "non-DRAFT" approaches. Having already considered that, in spite of its simplicity, the bus-based scheme still requires bus redundancy. Processor cost is a comparative measure that clearly does not favor DRAFT. Furthermore, because the bus has a fixed size regardless of the number of processors attached to it, we consider its cost negligible.

Let C represent the cost of a processor. We will first consider comparing DRAFT with a bus-based non-DRAFT approach. In this case all processor faults can be considered except Byzantine faults. Tolerating f faulty processors requires $2f + 1$ processors. Protecting the bus requires fail-safe processor operation. Duplicating processors to form a module that can detect its own failures is often done in bus-based, fault-tolerant designs [43] [44] [45]. The resulting module pairs are thus indivisible and have the processing power of only a single processor. Employing fail-safe operation doubles the number of processors required to tolerate f faults. Let the quantity c represent the fraction of C required for the dedicated unit in DRAFT. Then the cost ratio of bus-based non-DRAFT to DRAFT is:

$$\begin{aligned}
\text{Cost Ratio}_{bus-based} &= \frac{\overbrace{2C(2f+1)}^{bus-based, Non-DRAFT}}{\underbrace{(C(2f+1)+cC)}_{DRAFT}} \\
&= \frac{2(2f+1)}{(2f+1)+c} \qquad (6.10)
\end{aligned}$$

Owing to the simplicity of the dedicated RLSU, controller and ROM when it is compared with a processor's complexity, their combined cost is cC where $cC < C$ for $c < 1$. RLSU cost must be nonzero, and we will not consider any design alternative where the voter cost equals or exceeds that of a baseline processor. Therefore, as a cost multiplier, $0 < c < 1$.

Figure 6.22 shows cost ratio curves using equation (6.10) for various values of c.

Concern over a faulty processor potentially disabling an interprocessor communication channel means that, to achieve fault tolerance bus-based designs require, a minimum of twice the number of processors as in the baseline design. Arbitrary faulty bus behavior due to a Byzantine fault can cause failure of the voting process. However, as the bus is a simple entity in comparison with a processor, the use of a broadcast protocol together with an assumed dependable medium assures consistent message transmission in a bus-based system. We therefore restrict our consideration of Byzantine faults to the processors themselves. While only a single channel connects a processor to the other

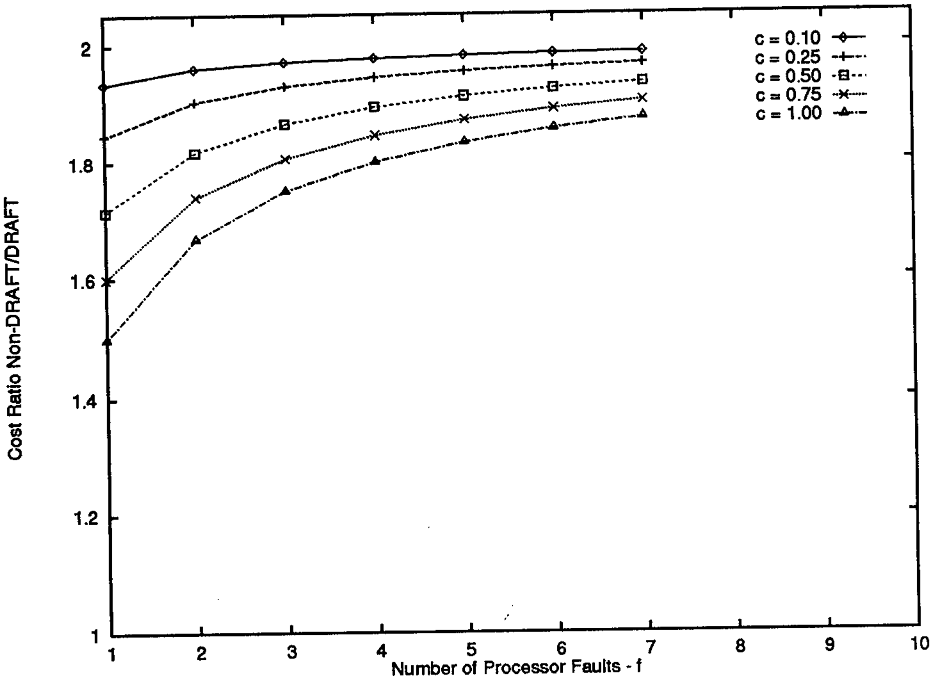


Figure 6.22: Cost Ratio of Bus-Based Non-DRAFT to DRAFT vs. Number of Processor Faults, *f*

processors in a bus-based design, in a point-to-point connection scheme a processor may have multiple channels that link it to other processors. Although we can assume that the message medium is totally reliable, a Byzantine fault can prevent a processor from simultaneously broadcasting a single message over its separate channels; instead, conflicting messages may be sent over each channel. As long as there are $2f+1$ disjoint communication paths from any processor participating in a vote to any other participant, enough correct information can be transferred to allow consensus to be reached in the presence of f Byzantine faults. Whereas the cost for connecting processors was considered negligible for the bus-based case, point-to-point connections means that a multiplicity of communication channels must be provided and the associated cost can no longer be ignored. Under the Byzantine fault assumption, the stipulation that there exist $2f+1$ *disjoint* paths between cooperating processors requires that each vertex in the graph representing processor interconnections have at least degree $2f+1$. Each edge in the graph corresponds to an independent communication channel that is shared by the two processors. With the required minimum of $3f+1$ processors (denoted by vertices in the graph), the total number of edges Num_{edges} is at least:

$$Num_{edges} = \lceil ((3f+1)(2f+1))/2 \rceil \tag{6.11}$$

The ceiling of the expression is taken because, if f is even, then the product $(3f+1)(2f+1)$ is odd; however, the total degree of all vertices in a graph must be even [46]. An even value for f means that there exists a vertex in the graph whose degree is even and greater than $2f+1$.

Equation (6.11) represents the total amount of processor interconnect. We could consider the cost of traces on a printed circuit board (PCB) that implement interconnect. Material costs aside, very often the power, weight, and size limitations determine the number of processors allowed in a system [3]; however this is in the context of non-Byzantine faults. The sheer physical complexity of processor connectivity for Byzantine fault tolerance can lead to systems whose resultant size is a reliability penalty [47] that can outweigh the benefits of fault tolerance. Added material costs would clearly be not just for extra PCB traces as the entire packaging concept for the system must change. Even if the increased system size is still acceptable, there are additional operational costs associated with the higher degree of connectivity. These include the additional mass of circuit boards made larger to accommodate interconnect area, and greater power supply demands caused by processors driving larger parasitic loads. Unlike the bus-based approach where each processor interfaces to a shared interconnection medium, multiple point-to-point connections to other processors impose specialized I/O interfaces on each processor.

We express the cost for interconnect as a fraction of a processor's cost. A single interconnection is assigned the cost $E \times C$ where E is interpreted as a fraction of a processor's cost. Then, for the point-to-point scheme, we have

$$\begin{aligned}\text{Cost Ratio}_{point-to-point} &= \frac{\overbrace{C((3f+1)+E \times Num_{edges})}^{point-to-point, Non-DRAFT}}{\underbrace{(C(2f+1)+cC)}_{DRAFT}} \\ &= \frac{(3f+1)+E \times Num_{edges}}{(2f+1)+c} \qquad (6.12)\end{aligned}$$

Using equation (6.12), Figure 6.23 shows the cost ratio curves for different values of E. In these curves, we assume the pessimistic, limiting value of $c = 1$ for the dedicated unit's cost.

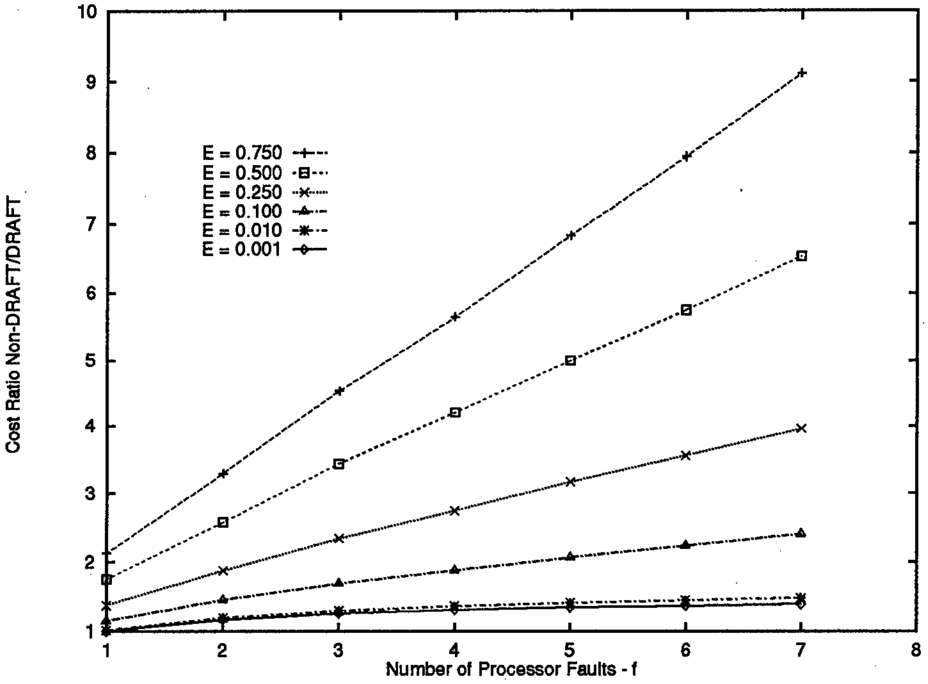


Figure 6.23: Cost Ratio of Point-to-Point, Non-DRAFT to DRAFT vs. Number of Processor Faults, f

The baseline DRAFT can tolerate up to $f = 2$ simultaneously faulty CMs. For equivalent fault coverage, the bus-based and point-to-point schemes have the costs shown in Figures 6.22 and 6.23 respectively. These figures show that, even when the cost parameters for the non-DRAFT options are given optimistic values, while those for DRAFT's dedicated unit are pessimistic, the cost advantage is still considerably in DRAFT's favor. In addition to DRAFT's cost savings is the previously pointed out improvement in application performance provided by DRAFT's dedicated unit. Next, we address extending the operable time of DRAFT beyond what is achievable with non-DRAFT approaches.

6.7 Extensibility of Operable Time

Operable time is the period during which a functional unit would yield correct results if it were being used. Reducing the probability of failure increases a system's expected operable time. The reliability function $R(t)$ is the probability that the system will perform satisfactorily from time zero to time t. The exponential distribution is one of the most commonly encountered in reliability models [65]. Using it as the fault distribution model together with a constant failure rate λ, then each processor has reliability

$$R_p = e^{-\lambda t} \tag{6.13}$$

A system's Mean Time to Failure (MTTF) is given by [3]:

$$\text{MTTF} = \int_0^\infty R(t)dt$$

Just as we assume that *cost* and *complexity* are directly related, we also assume that complexity is related to *failure rate.* In Mil-Hdbk-217E, the failure rate of a component is not considered to increase linearly with its complexity (measured in gates or bits on the chip) as one might think; instead, the system failure rate decreases as the level of logic integration increases. If the *same level of integration* is assumed for the components involved, then r is the fraction of a processor's failure rate realized by the simpler dedicated unit. This being the case, then the reliability of the voter is:

$$R_v = e^{-r\lambda t} \tag{6.14}$$

In the ideal case, the voter in an NMR system would have perfect reliability. This corresponds to $R_v = 1$, implying that the dedicated unit that carries out the voting function has zero times the complexity of a CM. The MTTF of a single processor and the MTTF of a 5MR system that has a perfect voter are given by the following equations:

$$\begin{aligned}
\mathrm{MTTF}_{R_p} &= \int_0^\infty e^{-\lambda t} dt \\
&= \frac{1}{\lambda} \\
\mathrm{MTTF}_{5MR_{R_v=1}} &= \int_0^\infty (10e^{-3\lambda t} - 15e^{-4\lambda t} + 6e^{-5\lambda t}) dt \\
&= \frac{47}{60\lambda}
\end{aligned}$$

Obviously, the MTTF of the 5MR system is lower than the MTTF of the simplex system. However, the plots in Figure 6.24 show that for certain values of λt the 5MR system is more reliable. The crossover point occurs where the 5MR and simplex system have equal reliabilities. The MTTF for each system is the area under its respective reliability curve, but a better comparison parameter here is the achievable *mission time.* This is the time t_m at which the reliability of a system falls below a predefined level [3]. To realize the reliability benefits demonstrated by the 5MR curve in Figure 6.24, the mission time t_m must be selected so that λt_m corresponds to a point before the crossover.

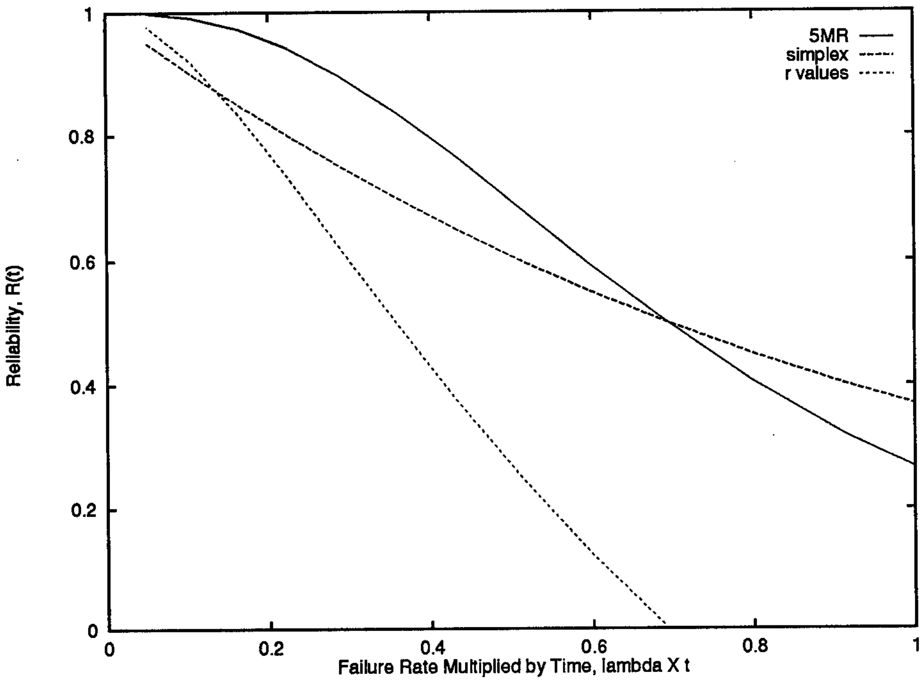


Figure 6.24: Simplex and 5MR reliabilities vs. λt

Our baseline DRAFT design has $N = 5$ CMs where 5MR would be the largest available fault-masking technique. We want to establish, in terms of processor complexity, how complex the voter can be and still yield a 5MR system that is more reliable than a simplex system. A 5MR configuration using five processors each with reliability R_p has reliability:

$$\begin{aligned} R_{5MR} &= (\sum_{i=3}^{5} \binom{5}{i} R_p^i (1 - R_p)^{n-i}) R_v \\ &= (10R_p^3 - 15R_p^4 + 6R_p^5) R_v \qquad (6.15) \end{aligned}$$

We wish to know when 5MR offers better reliability than the simplex approach, that is:

$$(10R_p^3 - 15R_p^4 + 6R_p^5)R_v > R_p \tag{6.16}$$

We determine the range of r values from the reliability constraints given in equation (6.16). Substituting equations (6.13) and (6.14) into equation (6.16) yields:

$$(10e^{-3\lambda t} - 15e^{-4\lambda t} + 6e^{-5\lambda t})e^{-r\lambda t} > e^{-\lambda t}$$

Solving this inequality for r yields:

$$r < 1 + \frac{\ln(10e^{-3\lambda t} - 15e^{-4\lambda t} + 6e^{-5\lambda t})}{\lambda t} \tag{6.17}$$

We use inequality (6.17) to show how the value of r affects the feasibility of system application for extensive mission lengths. Note that, because the 5MR reliability curve in Figure 6.24 assumes a perfect voter, it shows the maximum reliability that is obtainable. Because we are measuring complexity in terms of gate count, we know that $r = 0$ is a physical impossibility. The unit carrying out the voting function has a non-zero failure rate so it reduces the 5MR's reliability from that shown in Figure 6.24. From inequality (6.17) the values of r are found that, when multiplied by λt, would make the respective reliabilities for 5MR and simplex systems the same. The r values are also shown in Figure 6.24.

As r varies, the crossover points for the 5MR and simplex reliability curves change. For values of r that are close to zero, the crossover point is close to the one shown in Figure 6.24, but as r increases the crossover points occur sooner. This is shown in Figure 6.25 for three nonzero values of r.

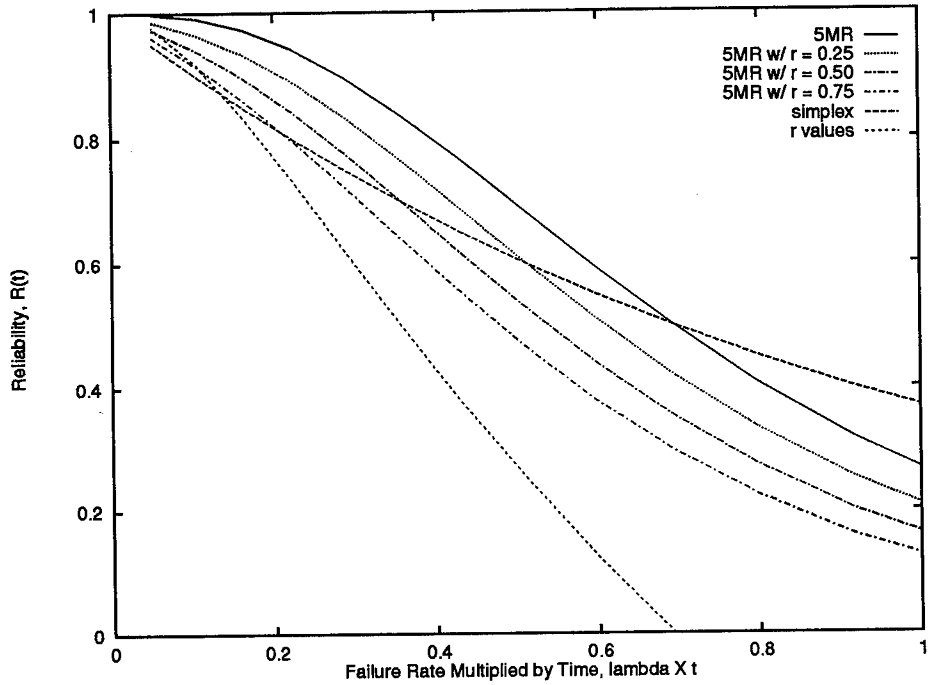


Figure 6.25: Simplex and 5MR reliabilities vs. λt

The extent to which operable times can be assured depends on r. Figure 6.25 shows that the smaller the r value is, the longer 5MR is more advantageous than simplex. Consequently, the user can select a longer mission time t_m as the complexity of the voting is lessened. The RLSU, RLSU controller, and memory for configuration data comprise DRAFT's dedicated unit. Because of the RLSU's self-reconfiguring capability, it requires only simple support devices: the RLSU controller is a simple counter and the external memory for configuration data is a ROM. The RLSU's logic resources are not static; instead they are used over and over again where a particular instance of RLSU functionality is derived from the ROM data. Initially there is a sharp expenditure in RLSU complexity because it is reconfigurable hardware. However, the requisite complexity for high functionality resides in mask-programmed ROM — these devices are highly dense and have the lowest failure rate of any type of electronic digital device [71]. Based on these considerations $r < 1$. In Chapter 7, we quantify the complexities of a processor and the RLSU, and show that for high-performance processors we have $r << 1$. Next we address whether the scalability of the dedicated unit is an overriding factor.

6.8 Scalability

The ability to increase the number of processors used in a DRAFT implementation depends upon how easily the architecture can be scaled. The component interconnections in DRAFT are the metal conductor buses prevalent in most electronic systems today. Maximum clock speeds for these buses is between 200 MHz and 300 MHz, whereas processor clock rates themselves are approaching those of the bus. This limits bus-based multiprocessors to support only a few processors because higher numbers of processors cause performance degradation as active processors saturate the bus. DRAFT is designed to provide high bandwidth architectural support for sharing and control. As opposed to the global shared memory often employed with bus-connected multiprocessing, the dual-ported RMs in DRAFT are private to each CM. Distributing the memory among the processors reduces memory contention to only those necessary instances when process coordination for access to the shared mailbox resource is required. The high performance of the RLSU mitigates the congestion of a centralized transfer point for messages and data. System clock rates for the RLSU can easily meet the minimum RM access times for both read and write operations.

Difficulties related to interconnect are encountered when scaling up a point-to-point multiprocessor scheme. As discussed in Section 6.6, for the system to be capable of tolerating f processor faults requires that there exist at least $2f + 1$ disjoint paths between any two processors participating in a vote. DRAFT allows *any* subset of available CMs to form a fault-tolerant processor. Knowledge of which processors will be available depends on the application load, and so it is a runtime issue. This uncertainty means that $2f + 1$ disjoint paths must exist between *all* processors of the multiprocessor if it is to offer the same flexible fault-tolerant operation as DRAFT. Our cost analysis for the point-to-point interconnection scheme shows that this connectivity becomes prohibitive for even a small number of processors when fault tolerance is required: given N processors, the required connectivity grows as N^2. Applying a fault-tolerance requirement (in this case, connectivity) to all processors leads to poor scalability using a traditional approach.

In contrast, universal application of a fault tolerance requirement to allow any subset of the processors to form a fault-tolerant processor *is* a feature of the DRAFT approach discussed here. In the baseline design, a 5MR voter is configured to vote on any subset of inputs or to perform a comparison of any two inputs. We address the scalability of DRAFT by considering its impact on configurable voting. Majority voting is widely used to mask hardware faults, and the complexity of its hardware implementation grows rapidly with N. The number of terms in the sum-of-products expression for the majority function of N inputs (where N is odd) is $\binom{N}{\lceil N/2 \rceil}$. Configuring an NMR voter is suitable for small N. Details of the sum-of-products implementation in the RLSU are shown in Chapter 7. Briefly, a horizontal collection of cells is used to form a product term. Buses run vertically between cell columns, and the value of a literal is carried by one bus per column so that each literal corresponds to a column. Therefore, the RLSU area, (in cells) needed to form the full sum-of-products expression for majority of N inputs is $N \times \binom{N}{\lceil N/2 \rceil}$. We contrast this with the following alternative approach that uses a routing network with the baseline 5MR voter as its core. The N ($N > 5$) inputs are grouped into non-disjoint subsets of size $N - 4$. Each element of a subset can be connected to a 5MR voter input. Figure 6.26 shows the formation of each subset and the allowable 5MR input for that subset.

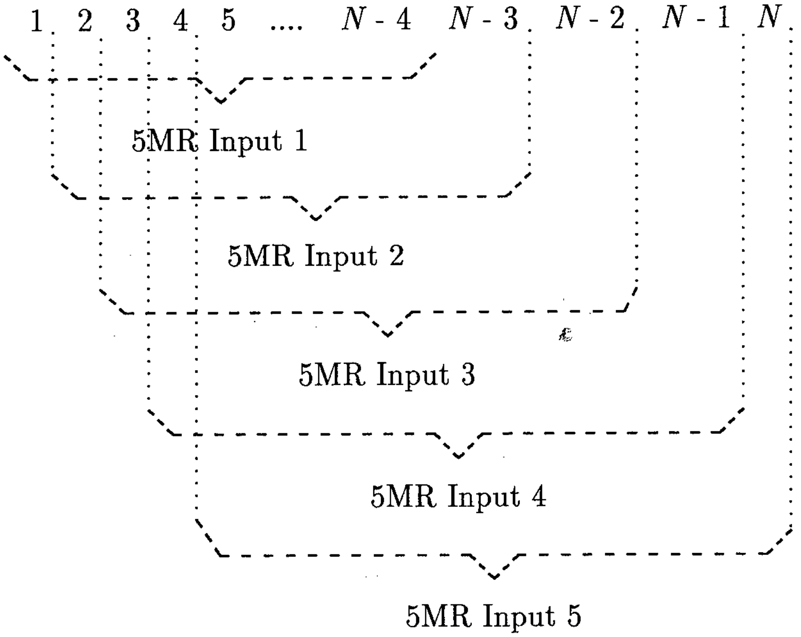


Figure 6.26: Grouping DRAFT Inputs

The configurable 5MR voter provided in the baseline DRAFT can now be applied simultaneously to any 5 of the N inputs. Detection signals are raised depending upon the nature of the inputs. Figure 6.27 shows the structure of the routing network to and from the 5MR core voter.

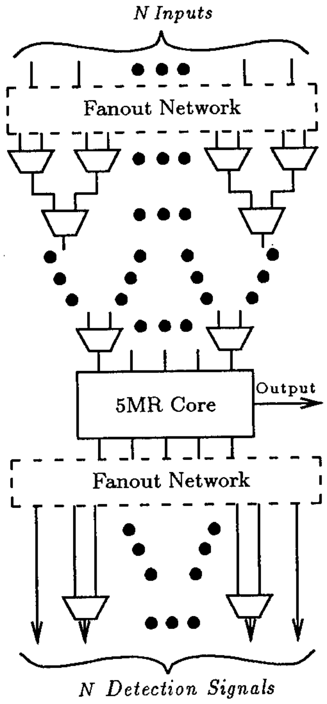


Figure 6.27: Scalable DRAFT

We now analyze the complexity, in RLSU cells, of the scalable design. The upstream fanout network in Figure 6.27 is approximately of height N to allow the vertical inputs to turn the corner and reach across to the multiplexer inputs. In Chapter 7 we show that two input multiplexers are readily available in the RLSU. The N inputs are grouped into 5 subsets. Routing of each subset to a voter input calls for $5 \times (N-4)$ cells of RLSU width. The multiplexer tree upstream from the 5MR core is of height $\lceil \log_2(N-4) \rceil$. As described earlier, the 5MR core is 10 cells high. Downstream is a 5 cell high fanout network that spreads the core's 5 detection signals across the width of a multiplexer tree. This multiplexer tree routes the core's detection signals to correspond to the appropriate processors. Its height is $\lceil \log_2(5) \rceil = 3$ cells.

The 5MR voter with upstream multiplexer tree and fanout network are provided for each bit of the inputs. However, the N detection signals are only 1 bit each, so the cost of the downstream fanout network and multiplexer tree is amortized over the number of bits in each input.

Based on this analysis, this approach requires

$$5(N-4)(N + \lceil \log_2(N-4) \rceil + 10 + N + 3)$$

cells. Note that we have included the cell count for the N detection signals.

Figure 6.28 shows the full sum-of-products and routing network complexities on a log scale versus the number of processors (N).

Figure 6.28 demonstrates how increasing the redundancy used for fault-masking can quickly result in a voter with unmanageable complexity. As a result, practical voting applications rarely employ more than 7 processors [12] [13]. For $N \leq 8$, direct implementation of the full sum-of-products is possible, but for $N > 8$ it becomes necessary to use a routing scheme and apply an nMR voter ($n < N$) to any n of the N possible inputs. Comparison of RLSU sizes suggests differences in the number of cell propagation delays from input to output, so Figure 6.29 indicates the relative impact of increased N on RLSU speed.

In the full sum-of-products implementation, the critical path initially crosses N cells to form the first product term. This is then summed with each of the remaining $\binom{N}{\lceil N/2 \rceil} - 1$ terms where a cell delay is incurred per term. Applying the routing network

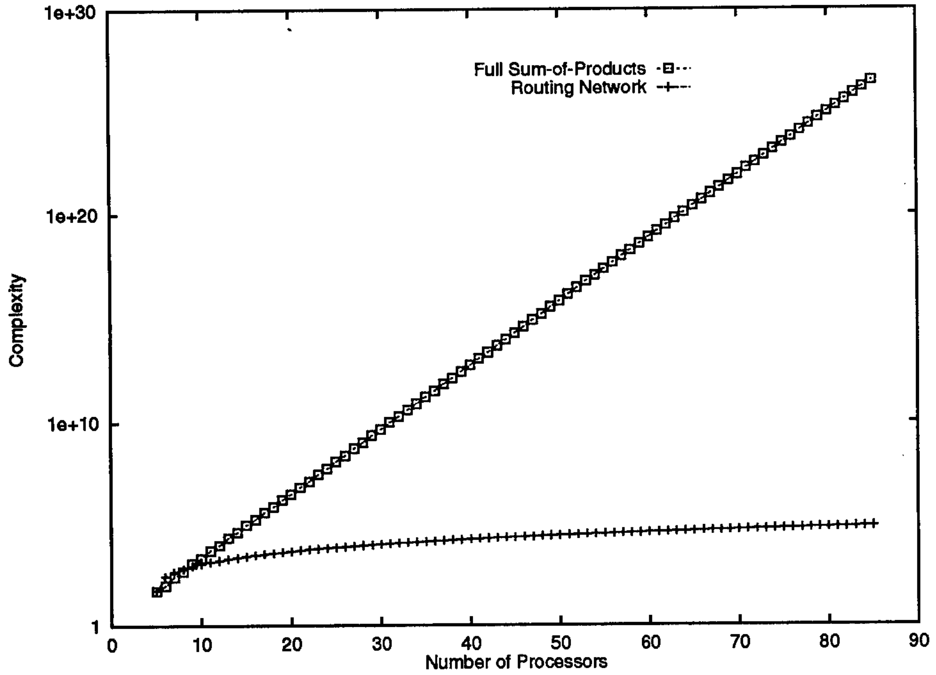


Figure 6.28: Full Sum-of-Product and Routing Network Complexities vs. Number of Processors (N)

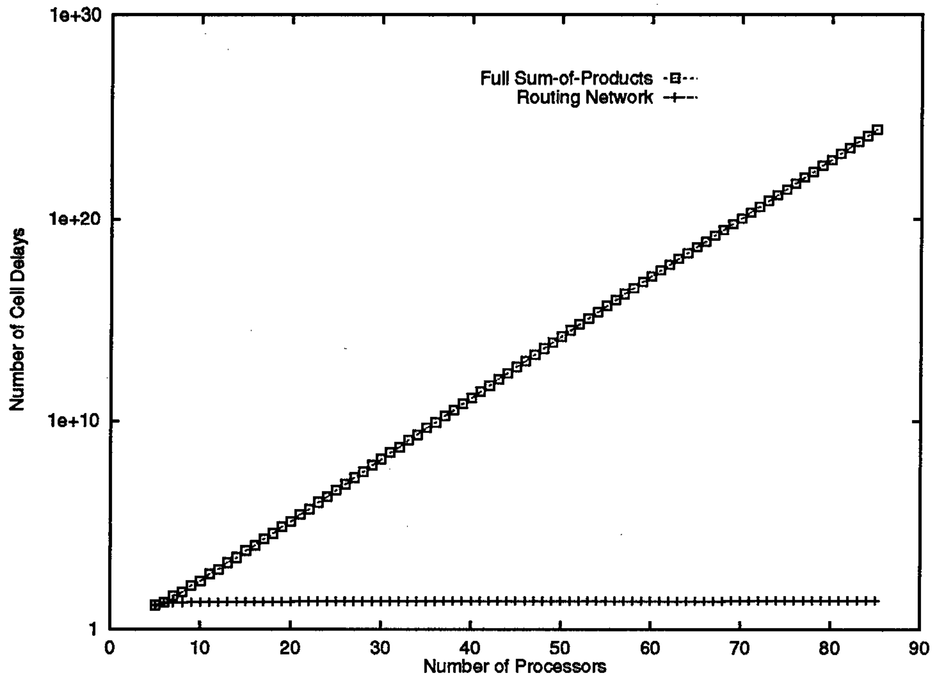


Figure 6.29: Full Sum-of-Product and Routing Network Cell Propagation Delays vs. Number of Processors (N)

to the critical path associated the sum-of-products of the 5MR core adds only $\lceil \log_2(N-4) \rceil + 3$ cell delays. Figure 6.29 shows the number of cell propagation delays for the full sum-of-products and the routing network on a log scale versus the number of processors (N).

6.9 Configurability

6.9.1 Processor Configurations

In this section, the flexibility of our proposed architecture is considered and the various configurations of the CMs in the baseline DRAFT are described. Previously, we have discussed the cases where a processor is in one of two operational modes: *fault-tolerant (FT)* or *multiprocessing (MP)*. A third mode is possible. It is when the processor is *unused (UU)*. A processor's operational mode is determined by the application assigned to it, whereas a processor is UU because either it is *idle* awaiting an application assignment, or it is in a *failed* state. Our objective is to create snapshots of DRAFT so as to determine all of the processor configurations where at least one CM is executing. We refer to these as the *active* configurations.

For the purpose of identifying combinations of modes, we do not distinguish between a processor in UU mode as being failed or idle. We take the view that an idle processor can experience a fault and therefore fail. Likewise, a processor that DRAFT has previously identified as failed could have since been repaired and replaced so that it can now be allocated. Also, although an MP application may be programmed according to either the functional or data parallel model, we do not need to make this distinction at this time in examining the processor configurations. When several processors are in MP mode, different configurations are possible. All the processors do not necessarily form a single multiprocessor. Recall that uniprocessing is simply MP mode with allocation of a single processor. Also, a set of processors in MP mode may be partitioned to form smaller multiprocessors. We maintain that hardware redundancy is necessary to tolerate processor faults, so at least 2 processors are required for an FT application.

We first consider the FT mode and the coexistence of processors in FT mode with others in the MP and UU modes. Then we will examine the MP mode together with the UU mode. Processors in the same mode may be grouped together in different ways. For instance, three processors in MP mode may be grouped together to form a three-processor multiprocessor, or a two-processor multiprocessor with a uniprocessor. Examples of FT mode are five processors forming a 5MR unit or a 3MR unit and a pair of processors that operate in the manner of duplication with comparison (DWC).

We use the notation $T_M(N, n, m)$ to denote the number of ways n processors in mode M can be used to create different configurations of the N available, and where m is the maximum number of processors among the n that can be grouped together. For instance, $T_{FT}(5, 5, 3)$ implies that there are $N = 5$ processors (as in the baseline design), and all of them (i.e., $n = 5$) are in FT mode, and at most three (i.e., $m = 3$) of the five can be grouped together. This gives the number of configurations where there is a 3MR unit and DWC unit. Some combined values of n and m are not possible, such as $T_{FT}(5, 5, 4)$. Once 4 processors are grouped together a single processor still remains, but that single processor cannot be used for an FT application because of the lack of redundancy.

Beginning with FT mode,

$$T_{FT}(5,5,5) = \binom{5}{5} = 1$$

When the maximum of 3 processors are grouped the remaining two must still form a fault-tolerant unit so they become a processor pair for DWC, which means

$$T_{FT}(5,5,3) = \binom{5}{3} = 10$$

Next we consider when four processors are in FT mode. Allocation of 4 processors to a single FT application means that there are $\binom{5}{4} = 5$ ways to group four out of the five processors and the remaining processor can be in either MP or UU mode, thus

$$T_{FT}(5,4,4) = \binom{5}{4} \times 2 = 5 \times 2 = 10$$

When two is the maximum group size when four processors are in FT mode there are $\binom{4}{2} = 6$ ways to form a DWC processor pair using the four available. However, the remaining two are in FT mode and they too must form a DWC processor pair, so only one-half (i.e., 3) are unique configurations. Thus

$$T_{FT}(5,4,2) = \binom{5}{4} \times 3 \times 2 = 5 \times 6 = 30$$

There are $\binom{5}{3} = 10$ ways to form a 3MR unit out of five processors. Each of the remaining two processors can be in MP or UU mode, yielding a total of four possible modes for both of them. Yet when both are in MP mode they can form either a two-processor multiprocessor or two independent uniprocessors, so there exists five possible configurations for these two processors. As a result

$$T_{FT}(5,3,3) = \binom{5}{3} \times 5 = 10 \times 5 = 50$$

There are $\binom{5}{2} = 10$ ways to select two processors from five. Three processors remain and we examine their possible configurations. Two modes are available for these processors — MP and UU. If all three processors are in MP mode they can form a three-processor multiprocessor or three uniprocessors (2 configurations). There are also $\binom{3}{2} = 3$ configurations involving two-processor multiprocessors and a uniprocessor. Among the three processors, if one of them is in MP mode, then there are $\binom{3}{1} = 3$ additional configurations. On the other hand, if two of the three processors are in MP mode, then there are $\binom{3}{2} \times 2 = 6$ configurations because the two processors in MP mode can be either a two-processor multiprocessor or two uniprocessors. Finally, all of the three processors can be in UU mode, so the total number of configurations for the three processors is $2+3+6+3+1 = 15$. Then

$$T_{FT}(5,2,2) = \binom{5}{2} \times 15 = 10 \times 15 = 150$$

Thus, for FT mode, the complete count is

$$T_{FT}(5,5,5)+T_{FT}(5,5,3)+T_{FT}(5,4,4)+T_{FT}(5,4,2)+T_{FT}(5,3,3)+T_{FT}(5,2,2) = 251$$

possible configurations.

Now we turn to configurations of processors in MP mode and UU mode. When all processors are in MP mode,

$$T_{MP}(5,5,5) = \binom{5}{5} = 1$$

For a uniprocessor and groupings of size four the remaining processors

$$T_{MP}(5,5,4) = \binom{5}{4} = 5$$

Owing to two processors forming either a two-processor multiprocessor or two uniprocessors, the number of ways to group three processors out of five is multiplied by two:

$$T_{MP}(5,5,3) = \binom{5}{3} \times 2 = 10 \times 2 = 20$$

There are $\binom{5}{2} = 10$ ways to select a two-processor multiprocessor from five available processors where the three remaining processors are uniprocessing. These three remaining processors have $\binom{3}{2} = 3$ configurations involving a two-processor multiprocessor as well. This does not, however, create an additional 10×3 configurations; instead only half are unique, so

$$T_{MP}(5,5,2) = \binom{5}{2} + \left[\binom{5}{2} \times \frac{\binom{3}{2}}{2}\right] = 10 + \frac{(10 \times 3)}{2} = 10 + 15 = 25$$

There is a single configuration where *all* the processors are uniprocessing, that is

$$T_{MP}(5,5,1) = \binom{5}{5} = 1$$

Next, when four of the five are in MP mode and the other is in UU mode we calculate

$$T_{MP}(5,4,4) = \binom{5}{4} \times \binom{4}{4} = 5 \times 1 = 5$$

$$T_{MP}(5,4,3) = \binom{5}{4} \times \binom{4}{3} = 5 \times 4 = 20$$

There are $\binom{4}{2} = 6$ ways to select a two-processor multiprocessor from four processors where the two remaining processors are uniprocessing. The two remaining processors could also form a two-processor multiprocessor. This does not, however, create an additional six configurations, because when they are paired and placed along with the previous pairs only three are unique configurations. With four processors in MP mode, the number of ways to select either a two-processor multiprocessor with two uniprocessors, or two two-processor multiprocessors, is $6+3=9$. Thus

$$T_{MP}(5,4,2) = \binom{5}{4} \times 9 = 5 \times 9 = 45$$

When all four processors are uniprocessing we have

$$T_{MP}(5,4,1) = \binom{5}{4} \times \binom{4}{4} = 5 \times 1 = 5$$

We turn to the situation where three of the five processors are in MP mode and the other two are in UU mode, to obtain:

$$T_{MP}(5,3,3) = \binom{5}{3} \times \binom{3}{3} = 10 \times 1 = 10$$

$$T_{MP}(5,3,2) = \binom{5}{3} \times \binom{3}{2} = 10 \times 3 = 30$$

$$T_{MP}(5,3,1) = \binom{5}{3} \times \binom{3}{3} = 10 \times 1 = 10$$

Next, when two of the five processors are in MP mode and the other two are in UU mode the situation is once again such that the two processors can form either a two-processor multiprocessor or two uniprocessors. So

$$T_{MP}(5,2,2) = \binom{5}{2} = 10$$

$$T_{MP}(5,2,1) = \binom{5}{2} = 10$$

Finally we consider when a single processor of the five is operating

$$T_{MP}(5,1,1) = \binom{5}{1} = 5$$

The total number of active MP configurations is 191. Thus, the total number of CM operating configurations for our DRAFT design is 442.

6.9.2 Analysis of an Alternative Design

We introduce an alternative architecture that can support FT and MP modes. Through our review of the relevant literature we believe that this approach is representative of the current state of the art in hardware-based selective fault tolerance. It will be shown how this architecture's lack of flexibility allows for far fewer processor configurations than in DRAFT, which translates directly into a large decrease in application processing throughput.

Figure 6.30 shows a five processor system consisting of processors P_1 through P_5, a voter V and a comparator C. This design recognizes the need for selective redundancy as not all processes are considered to be sufficiently critical to warrant triplicating their processors. A comparator is used with a pair of processors so that they can operate in the manner of DWC.

This type of system has been embodied in the AIPS fault-tolerant computer [17]. In the system depicted in Figure 6.30 we assume an *extended* capability by applying the concept of the C.vmp [4] to the processors at the sites of 3MR and DWC. That is, when they are not in FT mode the processors at each site can operate as either multiprocessors or multiple uniprocessors.

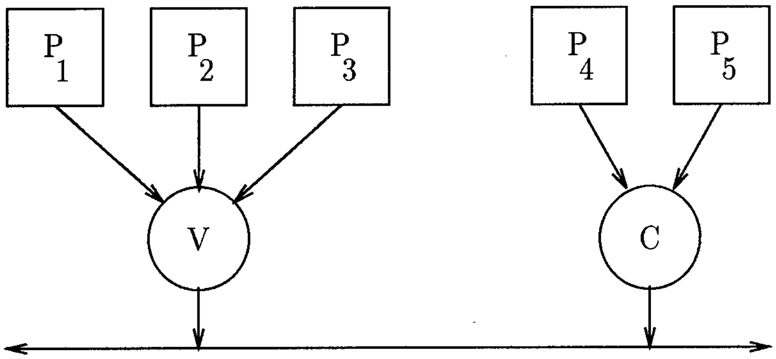


Figure 6.30: A Five Processor Comparative Architecture

For comparison with DRAFT, we calculate the number of processor configurations for the system shown in Figure 6.30. Previously, we calculated the number of possible DRAFT configurations of two and three processors when the available modes are MP or UU. For the two processor case there are five configurations, and for the three processor case there are 15. One possible configuration for the system shown in Figure 6.30 is when both the 3MR and DWC sites are in FT mode. Additional configurations exist when only one site is in FT mode. When the 3MR site is in FT mode, then the two processors at the DWC site can support five configurations. On the other hand, when just the DWC site is in FT mode, the three processors at the 3MR site can be configured in 15 different ways. Therefore, there are $1 + 5 + 15 = 21$ different active configurations involving FT mode. When neither site is in FT mode, then there are $15 \times 5 = 75$ possible configurations. However, one of these configurations has all processors in the UU mode, so after removing it from consideration the total number of active configurations for the system of Figure 6.30 is $21 + (75 - 1) = 95$.

DRAFT has 442 active configurations as compared to the 95 offered by the alternative design that has the same number of processors. Thus, DRAFT has over four times the number of alternative configurations.

It can be immediately seen that the alternate design does not support 5MR, 4MR, or multiprocessors of size greater than three. This places a tighter limit on the resource demands that an application can make. Previously, we categorized for DRAFT the supportable application streams based on the number of processors requested per application in a stream. Sets F and M have 15 and 31 elements respectively. Since we must restrict the number of processors an application can request in the alternate design to three, the sets are limited to $\{1\}$, $\{2\}$, $\{3\}$, $\{1,2\}$, $\{1,3\}$, and $\{2,3\}$ for MP and $\{2\}$, $\{3\}$, and $\{2,3\}$ for FT. Although the number of processors for both designs are the same, DRAFT supports 12 more types of FT application streams and 25 more types of MP application streams.

However, placing a limitation of three processor per application does not significantly diminish DRAFT's higher degree of configurability. From the 442 possible CM configurations we subtract those that involve 5MR, 4MR, or multiprocessors of size greater than three. That is

$$\begin{aligned} & 442 - (T_{FT}(5,5,5) + T_{FT}(5,4,4) + T_{MP}(5,5,5) + T_{MP}(5,5,4) + T_{MP}(5,4,4)) \\ = \quad & 442 - (1 + 10 + 1 + 5 + 5) = 420 \end{aligned}$$

unique configurations exist.

Applications arrive, execute, and complete at different times, so the configurations that we have enumerated for each system can be used to describe their activity. Nevertheless, given applications that can run on either design, the difference in the number of active configurations between the two systems is 440 − 95 = 325. The alternate design cannot be configured into any of these. Accounting for this difference is DRAFT's higher configurability where the only constraint on executing applications as they arrive is the existence of a sufficient number of available processors. The alternate system of Figure 6.30 lacks the flexibility of DRAFT. For example, an FT application requiring 3MR that arrives and finds a system state where one or two of the processors at the 3MR site have already been allocated cannot be dispatched. Even though there may be three available processors, this system structure specifically requires processors (P_1, P_2, and P_3) in order to provide 3MR. Moreover, the alternate design would never be in a configuration offering triplication where at least one of the processors at the 3MR site is in UU mode. Not only is it impossible to achieve triplication in this way, but if those processors in UU mode at the 3MR site are usable (i.e., not failed) then they would have been allocated instead.

Interpreting the UU mode also as the *unusable* mode instead of merely the unused mode, we can attribute the active configurations with at least one processor in UU mode to times when the system is operating with less than its full complement of processors because some have failed. These configurations represent situations where application support continues but with degraded throughput. DRAFT does not impose restrictions on which processors can be used, so degradation is more graceful. We have tabulated 325 active configurations that are unachievable by the alternate design. DRAFT, on the other hand, can enter these configurations because it sustains more combinations of concurrent applications and adapts more readily to processor failures.

By removing all restrictions on the arrangement of available processors before they can be used, resource utilization in DRAFT is maximized. Direct comparison with the resource utilization of the alternate approach would depend on the application stream statistics. However, resource utilization can be measured in terms other than processors. The ability of a system to meet changing circumstances will determine how well *any* of its resources can be used.

Processor failures adversely impact the ability of a system to maintain full support for its applications. By being unable to provide the resources for carrying out various application streams, the system will be underutilized. We quantify the extent of the impact through the tables shown in Figures 6.31 and 6.32 that show, for MP and FT respectively, the conditional probabilities of supporting application streams given that some number of processors have failed.

Application Request Set for MP Mode	Number of Failed Processors (f)							
	$f = 1$		$f = 2$		$f = 3$		$f = 4$	
	Alternate Design	DRAFT	Alternate Design	DRAFT	Alternate Design	DRAFT	Alternate Design	DRAFT
{1}	1	1	1	1	1	1	1	1
{2}	1	1	1	1	0.4	1	0	0
{3}	0.4	1	0.1	1	0	0	0	0
{1, 2}	1	1	1	1	0.4	1	0	0
{1, 3}	0.4	1	0.1	1	0	0	0	0
{2, 3}	0.4	1	0.1	1	0	0	0	0

Figure 6.31: Conditional Probabilities of MP Application Stream Support, Given f Failed Processors

Assuming that all processors have equal probability of failure, we calculated the conditional probabilities by finding the number of different configurations where f processors have failed and, of these, the number where the entire application stream can still be supported. Note that in the alternate design a processor failure at the 3MR

Application Request Set for FT Mode	Number of Failed Processors (f)							
	$f = 1$		$f = 2$		$f = 3$		$f = 4$	
	Alternate Design	DRAFT	Alternate Design	DRAFT	Alternate Design	DRAFT	Alternate Design	DRAFT
{2}	0.6	1	0.3	1	0.1	1	0	0
{3}	0.4	1	0.1	1	0	0	0	0
{2, 3}	0	1	0	1	0	0	0	0

Figure 6.32: Conditional Probabilities of FT Application Stream Support, Given f Failed Processors

site prevents triplication of processors. DWC would similarly be prevented by a processor failure at that site. Whereas a single processor failure in the alternate design immediately diminishes its fault-tolerant capability, 3MR and DWC are uninterrupted in DRAFT. DRAFT can experience two CM failures and continue to support processor triplication or duplication for both FT and MP. With probability 0.6, the alternate design is completely unusable for fault tolerance when two processors fail. On the other hand, with probability 1.0 three surviving CMs in DRAFT can support all MP/FT application streams.

The tables in Figures 6.31 and 6.32 show that as failed processors accumulate, DRAFT's application stream support diminishes much less rapidly. Due to its high degree of configurability, DRAFT is more resilient to changing circumstances by offering more opportunities for its continued utilization.

6.10 Programmability Support

The RLSU provides the routines for voting, comparison, and process coordination during access to shared mailboxes. Programmers can view the RLSU as a library of routines at their disposal. In our scheme, processes request RLSU functions in a manner similar to a function call. A function is specified in a mailbox message, and the function's parameters are included in the RM mailbox data. The RLSU returns the results to the calling process in the form of a message.

Although the functions are carried out in hardware, they are easily modifiable by altering the configuration code in ROM. If at a later time more functions are deemed necessary, then the contents of ROM would be augmented. Creation of the configuration code for a function is the outcome of the design phase for that function. Reconfigurable logic can dynamically alter hardware in real time, blurring the boundary between hardware and software. Programmability of the hardware itself is not the full story. Reconfigurable hardware technology has outpaced its own *design software* to the point where designers are unable to use dynamic reconfigurability to its best advantage [48]. Programming of tools for designing with dynamically reconfigurable logic was performed as part of this research — providing support for the creation of functions that execute at hardware speeds but whose implementation has the flexibility of software. In the next chapter, we describe a practical, highly-reliable, and relatively low-cost enabling technology for the RLSU and the development of design tools that are necessary to the successful application of this technology.

Chapter 7

Implementation – FPGAs

7.1 Logic Simulation

We turn to Field-Programmable Gate Arrays (FPGAs) as an enabling technology for the RLSU. Currently, the Computer-Aided Engineering (CAE) environments that are available for designing FPGA based systems do not support simulation of the FPGA reprogramming process, so prototyping of adaptive systems relies upon using the actual FPGAs. The FPGA architecture baselined in this section, similar to a commercially-available FPGA architecture, supports partial reconfiguration, to the individual cell level, without disturbing the rest of the array. In this section[1], we describe a modeling strategy for obtaining VHDL descriptions of versatile FPGAs so their dynamic behavior can be exhibited in advance of device procurement. An adaptive system using a versatile FPGA may also be prototyped with an emulation system whose FPGAs are architec-

[1] ©1995 IEEE. Reprinted, with permission, from *Proceedings of the Sixth IEEE International Workshop on Rapid System Prototyping,* Chapel Hill, North Carolina, June 7-9, 1995, pp. 174-180.

turally different from the one requiring emulation. VHDL structural descriptions of the prototype's FPGA demonstrate the feasibility of transferring the model to the emulation system. We show how the generation of both the model and the simulation input capture the FPGA's full versatility.

7.1.1 Logic Simulation of Dynamically Reconfigurable Systems

A highly configurable digital system adapts to various computational tasks through reuse of its available hardware for maximum performance with minimum overhead. Adaptive systems offer attractive features such as conservation of board space and the migration of algorithms from software to hardware. These systems can exploit reprogrammable FPGA technology by continually reconfiguring the existing hardware, and thus reduce the amount of hardware dedicated to system functions. Once the configuration data are read from an external memory, such as a PROM, and loaded into the FPGA, a function can be performed directly in hardware.

A major obstacle for the designers of adaptive systems to overcome is that current FPGAs are still too small when used singly. Another is that most FPGAs require complete reloading of the program memory when changing even a single programming bit, requiring them to be offline while they are programmed. To permit systems to become more adaptive, FPGA manufacturers are striving to produce devices that are more versatile. For example, Atmel offers FPGAs with a capability trademarked by the company as *Cache Logic* [49]. These devices can be partially reprogrammed without disturbing the rest of the array, and reprogramming can occur even during operation.

This permits continuous operation of an active function while another function is fetched from external memory and loaded into an inactive part of the chip. A large function can therefore be broken down into smaller functions so that some of these smaller functions are performed in the available hardware while the inactive functions are stored externally. Thus, the amount of hardware needed to implement a function can be significantly reduced.

Although FPGAs offer a widening range of opportunities for designers to make their digital systems adaptable, the available CAE environments for FPGAs do not support full exploitation of this capability. A survey of vendors' catalogues [49] [50] [51] revealed that none of the CAE tools for designing FPGAs permit the *simulation* of FPGA programming. Adaptive systems have potentially greater complexity due to their multi-mode nature and are thus strong candidates for rapid prototyping; yet to verify the system's reconfigurable behavior, designers must rely upon the actual target FPGA. Inevitably, breadboarding must be used to demonstrate FPGA reprogramming as part of system reconfiguration. This requires that the actual FPGA and its design system be procured prior to prototyping.

Simulation of FPGA reprogramming as part of adaptive computing would allow prototyping earlier in the design cycle, and, in addition, allows different FPGA architectures to be modeled and used in advance of device procurement. Without simulation of FPGA reprogramming, the debugging of dynamic device reconfiguration must be done entirely at the breadboarding stage. Simulation of FPGA reprogramming as part of adaptive computing would allow prototyping earlier in the design cycle where changes in the system design can be more easily accommodated. Simulation provides several benefits over breadboarding, especially in the most preliminary stages of design. A simulation provides information about uninitialized and unknown signals (i.e., X states) that the actual hardware cannot. A simulator can be used to trace internal signals that are inaccessible to physical probing. Later in the design process, fault simulation can be used to gauge the degree to which the test cases verify the correctness of the design [52]. In situations where simulation times are long, system verification could be sped up greatly by porting the FPGA design to an emulation system. These systems enable the exercising of digital designs in a real operating environment several orders of magnitude faster than a simulator. The prototype at this stage allows the software developers to write and debug their software in parallel with the detailed hardware development [53]. Emulation of a design is based on transferring its hardware description into a prototype consisting of FPGAs. Commercially-available emulation systems may use FPGAs that are architecturally different from the FPGA to be emulated, so a hardware description language version of the device is needed as system input. The VHSIC Hardware Description Language (VHDL) [54] has been adopted by the IEEE and is now an industry standard for digital designs.

This section describes an application of VHDL in modeling and simulating FPGA-based adaptive systems. In this method, a VHDL structural model of a versatile FPGA is automatically generated so that multiple configurations of the device are possible — even while the model performs a mission function. Support is provided for modeling the design at different levels of abstraction. As demonstrated in Section 7.2, adaptive computing applications are achievable by reprogramming only a few individual cells that are distributed throughout the array, so modeling the FPGA at the cell-level is required. A structural description is the simplest to synthesize into hardware [55], so our simulated, detailed VHDL model of the FPGA makes it ready for logic emulation.

In Subsection 7.1.2, we describe the baselined FPGA. Subsection 7.1.4 contains the approach taken to model the FPGA's versatility. VHDL simulations are described in Subsection 7.1.8. Concluding remarks are found in Subsection 7.1.9.

7.1.2 FPGA Cell Model

The FPGA implementation considered in this section is based loosely on the Atmel architecture [49]. In the Atmel FPGAs, each cell can perform a combinational function, a sequential function (a d-type flip-flop), or both. In addition to logic and storage, these cells can also be used as simple "wires" to connect cells over short distances. For fast communication over longer distances, buses run horizontally between rows of cells and vertically between columns of cells. In addition to bus connections, an Atmel FPGA cell receives two inputs (A and B) from, and provides two outputs to, each of its north, south, east, and west neighbors. Figure 7.1 shows the model of the cell architecture. At the top of the figure are two independently configurable multiplexers - one multiplexer for the A inputs and another for the B inputs. Each of these multiplexers can also select a logical constant 1 so as to provide a constant 1 or 0 at outputs of the cell. The two four-input multiplexers providing the cell's outputs are controlled in tandem. In the Figure 7.1, the network composed of configurable bus drivers and supporting program enable logic provides read/write access to four buses: north-south-1 (ns1), east-west-1 (ew1), north-south-2 (ns2), and east-west-2 (ew2).

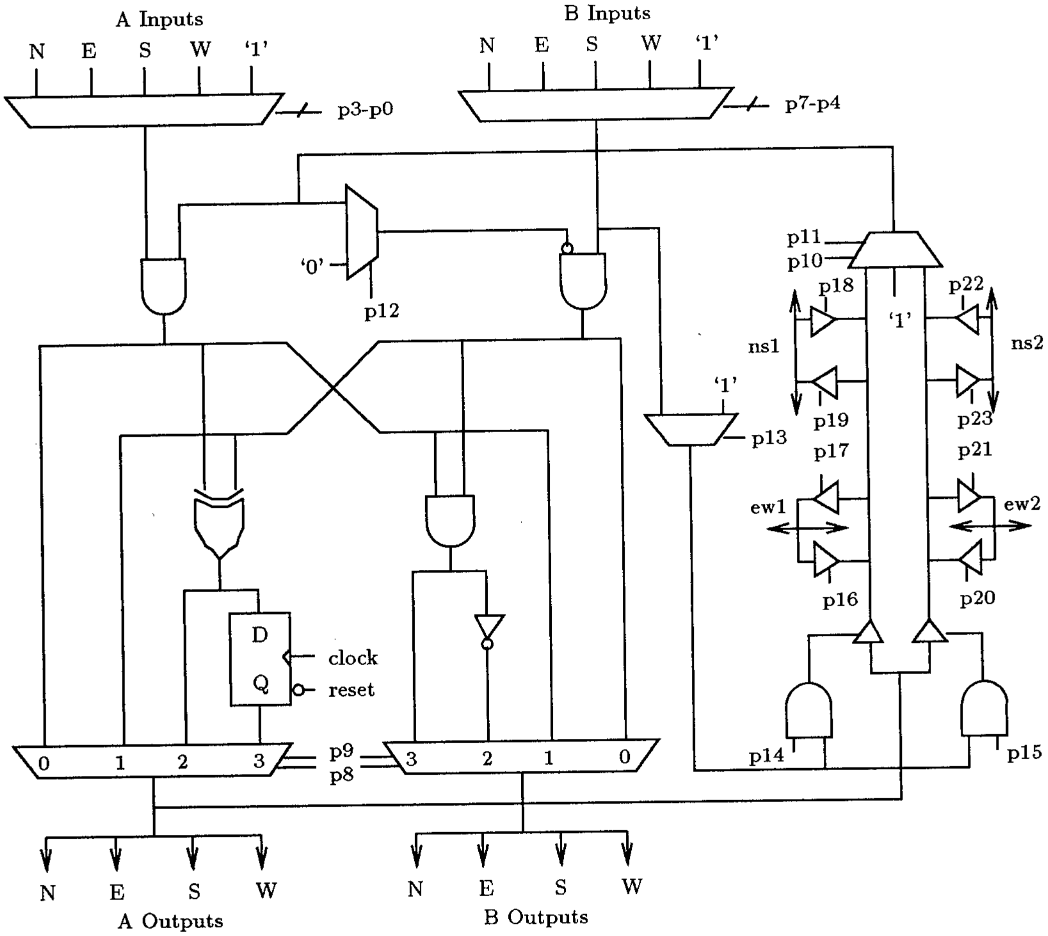


Figure 7.1: Cell Architecture

7.1.3 FPGA Programming

Although several programming methods may be available for a given FPGA, a method common to all reprogrammable FPGAs is the serial loading of programming data into the device. This is the only programming method considered in this section. Figure 7.2

shows a model of the FPGA programming logic and a cell's program memory. There are FPGA inputs for program data, a clock, program enable, and a load signal. These inputs, together with two decoders - row and column number - comprise the programming logic.

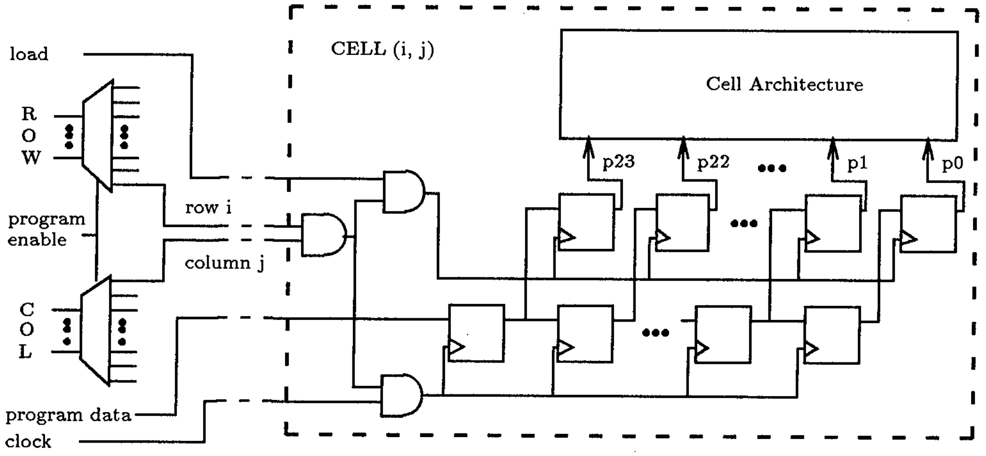


Figure 7.2: FPGA Programming Logic and Cell Program Memory

Associated with each cell are two registers: a shift register to accept the serial data and a parallel-load register to hold the program data for the cell. As previously mentioned, a feature of Atmel's FPGAs is the ability to reprogram any cell without disturbing the rest of the array; our FPGA model implements this feature by making individual cells addressable for programming. The shift register of an addressed cell is enabled and serially loaded with the cell's programming data. The shift register's contents are then transferred in parallel to the hold register.

7.1.4 Initial Design Description Format

The *Netlist Intermediate Form (NIF)* is a Rome Laboratory-developed language for transfer of digital designs [56]. NIF contains only structural information; no behavioral or functional data are included. The wide variety of CAE workstations produce an equally wide variety of netlist outputs, and many design verification techniques have not been implemented using these netlist formats. In response to this need, NIF is the target language for a set of translators that accept existing simulation languages as input, and it is the source language for a set of translators that produce existing simulation languages as output. This is far more efficient than writing a direct translator from every language of interest to every other. Other languages have been developed expressly for the purpose of transferring design data. For example, the Electronic Design Interchange Format (EDIF) [57] allows a user to describe many types of data about a design. NIF's description of gate interconnections is more convenient than EDIF's, and EDIF-to-NIF and NIF-to-EDIF translators have already been built. An advantage of using NIF was that the FPGA models could also be translated into a hardware description language (such as the Navy's HITS [58]) that supports fault simulation.

We describe the use of NIF in handling purely structural descriptions of designs at the gate or block level. A *model* contains *macros* (for hierarchical descriptions) and a *main model.* Macros and the main model have the same syntax. The building blocks are *components*, which each have a *name*, *type*, *inputs*, and *outputs.* Inputs and outputs are lists of signals. A component type is either the name of an already-defined macro, or it is assumed to be a *primitive* (a primitive can be a NAND gate, for instance). The

NIF does not interpret or give meaning to primitives — that is solely the task of the translators. Figure 7.3 shows a hierarchical schematic for a small network. The name of the macro is "SUB," and "DNDL" is a defined in the target simulation language.

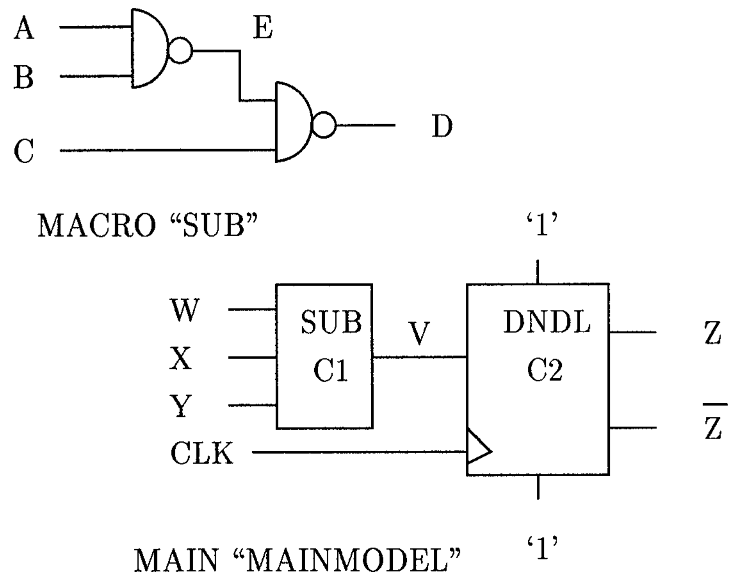
7.1.5 FPGA Generation

An FPGA model consists of the following components: cells, buses, row-decode, and column-decode. Figure 7.4 shows five cells of an FPGA. Each cell is identified by its row and column coordinates. CELL2_2 is connected to four adjacent cells and buses. Only CELL2_2's connections to the other cells are shown. The east-west buses are shared by cells in the same row, and north-south buses are shared by cells in the same column. Figure 7.4 shows the bus labels prefixed with their respective row or column number.

The regular structure of the FPGA facilitates automatic construction of the NIF model. A program was written to create the NIF model for an FPGA of any desired size. In the model, each component cell's A, B, and four bus outputs are labeled using the cell's coordinates as a prefix (e.g. CELL1_2 produces outputs 1_2A, 1_2B, 1_2NS1o, 1_2NS2o, 1_2EW1o, and 1_EW2o). The connections to a cell are determined by the cell's coordinates. For example, component CELL2_2 appears in NIF as shown in Figure 7.5.

Type CELL is a macro made up of gates, muxes, 3-state buffers, and a flip-flop as shown in Figure 7.1. A bus component, such as ns1 in column 2, appears as shown in Figure 7.6.

FPGAs require that the cells and interconnect be programmed prior to use, so an FPGA programming and application language was developed along with an interpreter



```
MODEL (
        MACRO (SUB
                C (Y NAND IN (A B) OUT (E) )
                C (X NAND IN (C E) OUT (D) )
        INPUT (A B C)
        OUTPUT (D)
              )
        MAIN (MAINMODEL
              C (C1 SUB IN (W X Y) OUT (V) )
              C (C2 DNDL IN (V CLK '1' '1') OUT (Z ZBAR) )
        INPUT (W X Y CLK)
        OUTPUT (Z)
              )
        )
```

Figure 7.3: Sample Network and its NIF description

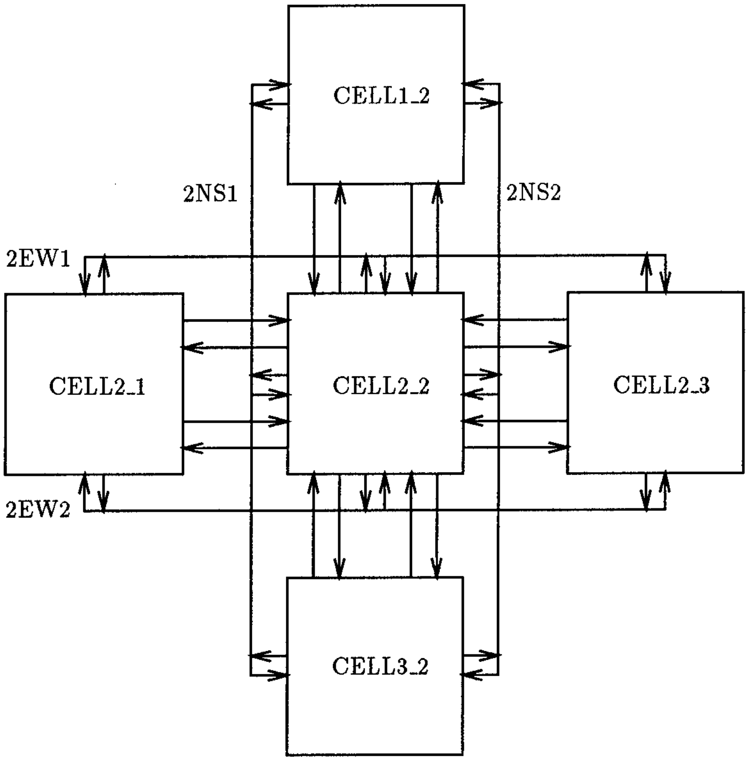


Figure 7.4: Cell Interconnections

```
C (CELL2_2 CELL   IN     ( 1_2A 1_2B 3_2A 3_2B
                           2_3A 2_3B 2_1A 2_1B
                           2NS1 2NS2
                           2EW1 2EW2
                           clock reset load
                           row_i column_j
                           program_data )
                  OUT    ( 2_2A 2_2B
                           2_2NS1o 2_2NS2o
                           2_2EW1o 2_2EW2o ) )
```

Figure 7.5: NIF Component Declaration of CELL2_2

```
C (2NS1 BUS   IN     ( 1_2NS1o 2_2NS1o 3_2NS1o )
              OUT    ( 2NS1 ) )
```

Figure 7.6: NIF Component Declaration of a Bus

that converts this language into the necessary simulation input. The interpreter can also put the FPGA program data into an appropriate format for PROM storage. It also simultaneously creates the NIF model of the FPGA and its programming data.

To reduce both the model complexity and the simulation times, the unused cells were modified. An unused cell can be greatly simplified because the only path for its outputs to a primary output is through the bus interface, which is disabled. As a result, an FPGA cell unspecified in the programming language is replaced by a *stub* of a full cell. Note that only cells that are unused in *every* configuration (during a single simulation) can be made stubs. All programming logic was removed from a stub and its architecture was reduced to only disabled bus drivers and a single gate that sinks all the cell's inputs and sources a constant output of zero. For example, if CELL2_2 were unused, it would appear in the NIF model as shown in Figure 7.7.

```
C (CELL2_2 STUB   IN     ( '0' '0' '0' '0' '0' '0' '0'
                           '0' '0' '0' '0' '0' '0' '0'
                           '0' '0' '0' '0' )
                  OUT    ( 2_2A 2_2B
                           2_2NS1o 2_2NS2o
                           2_2EW1o 2_2EW2o ) )
```

Figure 7.7: NIF Component Declaration of a Stub

Cells can be programmed by either specifying a standard state (combinational, sequential, or constant), or by individually defining each programming bit. A programming language example appears in Figure 7.8.

In this example, the interpreter will generate the vectors to set CELL2_2's programming bits for the standard (i.e. predefined) combinational state "cs20." This state calls for the cell to be configured as a two-input (A & B) multiplexer with output selection

```
prog =   (2,2)
         type     = (std)
         state    = (cs20)
         input    = (AN, BE, NS1)
         output   = (EW2)
```

Figure 7.8: FPGA Programming Language Example

provided by a bus. The cell's A output is the multiplexer output. Bits p8 and p9 (see Figure 7.1) are set to zero, and p12 is programmed so that the bus input will control selection of either A or B. For its inputs, CELL2_2 uses the output of its northern and eastern neighbors for A and B respectively. Determining the selection for the cell's multiplexer function is bus ns1. Fanout of the output is always to the adjacent cells; however, the A output in this example is also routed to ew2 bus.

The individual setting of program bits is done when bus connections between cells must "turn corners," say, from east-west to north-south.

Data vectors that exercise the operation of the programmed FPGA also appear in this programming language. When the interpreter encounters them, they are formatted for application to the FPGA's primary inputs. During initial programming of the device, "don't care" values (Xs) are placed on the data inputs. Thereafter, data vectors can be flagged so they will be simultaneously applied with the serial program data.

7.1.6 Versatile Reconfiguration

Figure 7.9 shows how a versatile FPGA could be used to implement multiple functions. In this example, four different functions are successively performed on the input data. The two-input multiplexer is implemented using a single cell as described in Subsection 7.1.5. A collection of cells occupying area a_i of the FPGA is sufficiently large enough to implement functions f_1 or f_3, where f_1 is an adder and f_3 is a subtractor. Similarly, area a_j can hold either f_2 or f_4 where these functions carry out, respectively, comparison of an input value against an upper and lower bound. As only one function is needed at a time, the area occupied by the other function is inactive. Loading of the next function's program data into this area occurs without interrupting the operations of the current function.

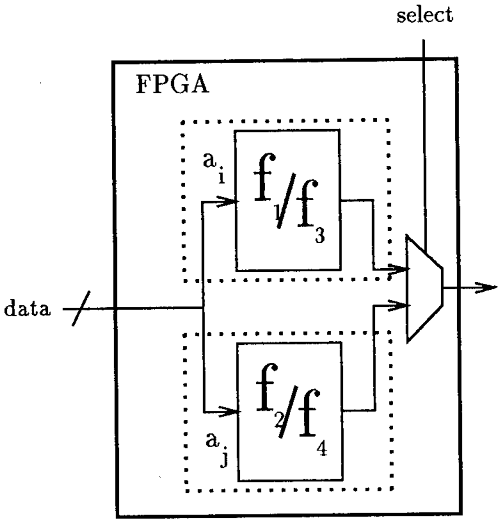


Figure 7.9: Parallel Operation and Programming

In support of the concurrent programming and functional operation described above, the FPGA programming language steps have the following form:

1. Program cells of a_i for f_1.

2. Program cells of a_j for f_2 while applying data vectors.

3. Program cells of a_i for f_3 while applying data vectors.

4. Program cells of a_j for f_4 while applying data vectors.

We have described the generation of both the model for the versatile FPGA and its programming input. Seamlessly merging data inputs with the programming inputs demonstrates reconfiguration in parallel with operation of the mission functions. In the above example, all four functions are "available" in hardware; however, only two are embodied by the FPGA at any time. Next, we discuss the transfer of this design data to VHDL.

7.1.7 Translation to VHDL

The translation from NIF to VHDL was straightforward because the FPGA models created were purely structural. The VHDL models were based on a four-valued logic (0, 1, X, Z).

VHDL has no built-in primitive gates, such as AND and OR; these primitives must be implemented behaviorally. A gate-level library was used to provide the required "primitive" gates, which included bus drivers and flip-flops.

The NIF-to-VHDL translator automatically creates the VHDL code that is normally superfluous to the user, such as the signal and component declarations. Component instantiations were mapped one-for-one from the NIF model. The precompiled gate-level library does not usually contain every gate type needed for every logic model so the translator also creates the behavioral code for the missing gate types.

The VHDL simulation environment has a cumbersome interface to the outside world.

The translator creates a customized VHDL test bench that reads ASCII vectors and writes the stimulus/response data as an ASCII file. This permits the same test vectors to be used for emulation in VHDL as well as be fault-graded in another hardware description language such as HITS. Furthermore, these input/output data formats are compatible with a wide array of vector preprocessors and postprocessors developed at the Rome Laboratory.

7.1.8 VHDL Simulations

Once translated to VHDL, the FPGA model was simulated in the Model Technology, Inc., VHDL environment. Data vectors in these simulations were also combined with programming bits to verify the dynamic reprogrammability of the device — even while it is operating.

While it is useful to simulate only the FPGA, reprogramming a FPGA when it is in a system requires support chips. Figure 7.10 shows the high-level schematic of a VHDL model of a board translated from NIF.

The PROM and FPGA Controller are components declared in NIF but whose behavior is defined in a VHDL library. The FPGA component is a NIF macro composed of many other macros and translated to a detailed structural VHDL model. The PROM contains the programming data for implementing those functions that the FPGA must embody. Together, these components make up a basic FPGA-based adaptive system.

This model was used to simulate the *in situ* programming of the FPGA as described in Subsection 7.1.6. A state machine in the FPGA controller coordinates FPGA pro-

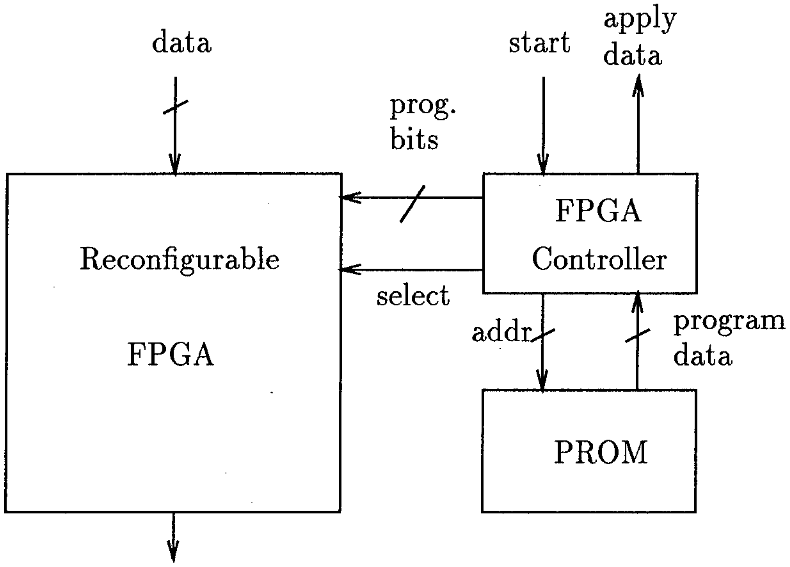


Figure 7.10: Basic Adaptive System

gramming and execution. The controller receives a signal from the VHDL test bench to initiate FPGA programming. The test bench applies data vectors to the model only when signaled to do so by the controller. Simulations have been performed that demonstrate reprogramming the FPGA with a new function, while it simultaneously performs an operation on the data vectors applied to its inputs. From this model of a basic adaptive system, we constructed the design for the Reconfigurable Logic and Switching Unit (RLSU) described in Chapter 4. There we described the RLSU as self-reconfiguring, so that after the major control functions are migrated to the FPGA itself, all but a counter remains as the FPGA controller.

The PROM and controller of Figure 7.10 are chip-level primitives in that they are not constructed hierarchically from more basic primitives. The FPGA we have described is a hierarchical model. These different model types were connected through NIF to create our simulatable adaptive system. Because simulating the low-level FPGA model could be prohibitively expensive, a behavioral description of the cell was written. Exhaustive simulation of the behavioral cell verified that its output responses were identical to those of its structural version. By simply deleting the cell macro from the NIF model and inserting the cell's corresponding behavioral code in the VHDL library, a more abstract FPGA model was generated in which every structural cell was replaced with its behavioral counterpart. This resulted in a decrease in the simulation times for the adaptive system, and an increase in the size of the FPGA models that the simulator could support. Carrying the abstraction process further, collections of cells that perform various FPGA functions (e.g., f_1, f_2, f_3, and f_4 of Subsection 7.1.6) were given behavioral descriptions and connected, using NIF, to ultimately produce a behavioral VHDL description for the entire FPGA. When simulation becomes too slow, even at this highest level of abstraction, the structural VHDL FPGA model allows transfer to an emulation system.

7.1.9 Summary

Adaptive systems are possible through FPGA technology. Until now, however, the only way to demonstrate the capabilities of these systems is to use the actual FPGAs. To prototype an adaptive system more rapidly, we proposed a strategy to model versatile FPGAs. This strategy has been applied in the transfer of an FPGA design to VHDL. This same strategy can be applied to create VHDL models of other FPGA architectures.

Simulations of the FPGA both as a stand-alone device and in concert with VHDL models of a PROM and controller were performed. In both types of simulation, the FPGA's full versatility was demonstrated. An automated process creates the VHDL description of the FPGA for simulation at different levels of abstraction, and, if necessary, for transfer of the model to an emulation system.

7.2 FPGA Layouts

We now show how the FPGA described in Section 7.1 can be rapidly reconfigured to support all of the options available in both the fault-tolerant and non-fault-tolerant modes.

FPGA reconfiguration service has to be provided for task output. FPGA routing of non-fault-tolerant task output is to the host, the MC, or to other tasks. When applying fault-tolerance across subsets of the N computing modules, FPGA reconfigurations are required because a subset may be in the form of passive redundancy (from any 3 modules up to N modules) or active redundancy (any 2 modules).

All the modules have physical connections to the FPGA, but, unless a task is redundantly executing on all the modules, it uses only some of these connections. The other connections still carry some logical value, and if these values are used by the FPGA, an erroneous output could result. If less than NMR is called for, a modification to the FPGA is necessary; but it is our goal to preserve as much of the current FPGA configuration as possible so only small differences exist among all the necessary configurations. In an Atmel FPGA consisting of up to 6400 cells, the time to program any individual cell is 0.2 μs [49]. Small differences between any two configurations mean that only a short time is required to switch between operations. Thus, incremental programming boosts throughput. Also, the need to store only small differences lessens the amount of PROM storage needed for configuration data.

7.2.1 N-Modular Redundancy Example

Consider, as an example, a voter designed to produce as output the majority of 5 modules. Because the majority vote function employs passive redundancy, errors are masked. However, with the addition of a bank of XOR gates, module errors are revealed by comparing the majority with each voter input. Figure 7.11 shows how a circuit can be changed so that it functions as either a 5MR or 3MR voter. The gates in this example are unusual in that they can be programmed to output a constant 1 or 0. For 5MR, the squares in Figure 7.11 represent logic gates programmed so that their input is passed directly to their output. When switching to a 3MR vote, the sum-of-products is appropriately reduced by programming some gates to output a fixed 0 or 1 as illustrated by arrows in Figure 7.11. For this example, the output expression for 5MR is reduced to $bc + bd + cd$ when error detection for inputs a and e is disabled. This expression now determines the majority of only the inputs b, c, and d. Other combinations of 3 inputs can be similarly obtained.

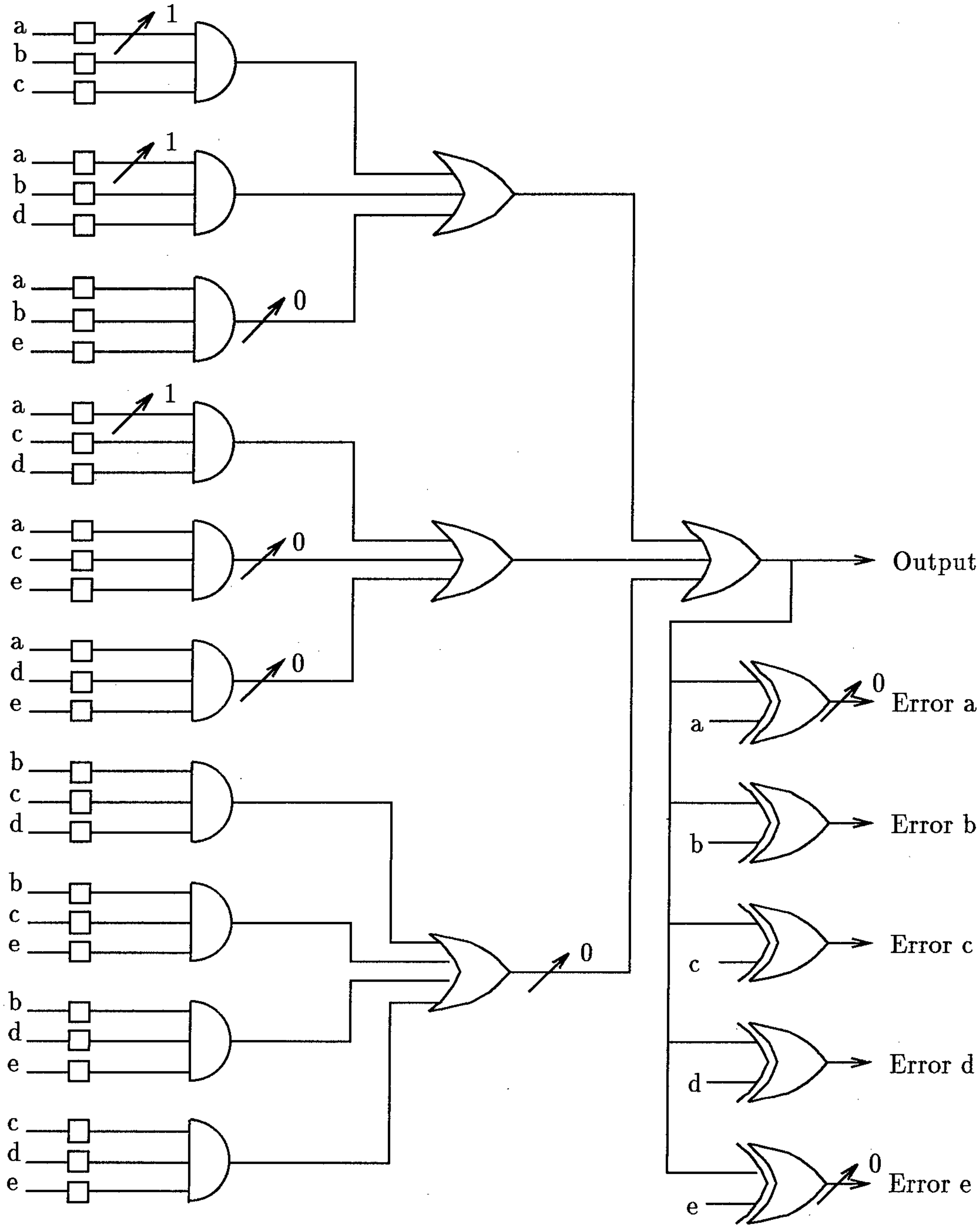


Figure 7.11: 5MR Gate-Level Voter

7.2.2 FPGA Design

Figure 7.12 shows a bit slice of the FPGA design that supports NMR, nMR (where $n < N$) and duplication with comparison. The darkened squares are bidirectional I/O pads, and the undarkened squares are logic cells. In this representative layout, five modules (a through e) are connected to the top I/O, and the MC is connected to the bottom I/O. Some cells are used for routing, while others are shown performing a logic function. The symbol "$\bullet$" in a cell indicates that the cell performs an AND function, a "$+$" indicates an OR, and a "$\oplus$" indicates a XOR. Arrows indicate the direction of signal flow. Figure 7.12 forms the core FPGA design — all other necessary configurations are derived from this design.

In describing the FPGA design, individual cells are identified by their row and column coordinates, indexed from 1, with $cell(1,1)$ being the cell at the top left corner. The six cells $cell(15,1)$ through $cell(15,6)$ are used for routing any single FPGA input to the output. In the core design, the first five of these cells are each configured to pass their input to their eastern neighbor, while the sixth cell passes its northern input to its output. Only $cell(15,6)$ and one more cell require reconfiguration for routing either a, b, c, d, or e to the FPGA output. For example, routing input c is accomplished by programming $cell(15,3)$ to accept c from the bus instead of from the output of its western neighbor. Now c flows to the input of $cell(15,6)$ which has been programmed to pass to its output its western input instead of its northern input.

The submatix of cells from row 5 to row 17 and from column 1 to column 6 perform the sum-of-products majority function with disagreement detection. The majority out-

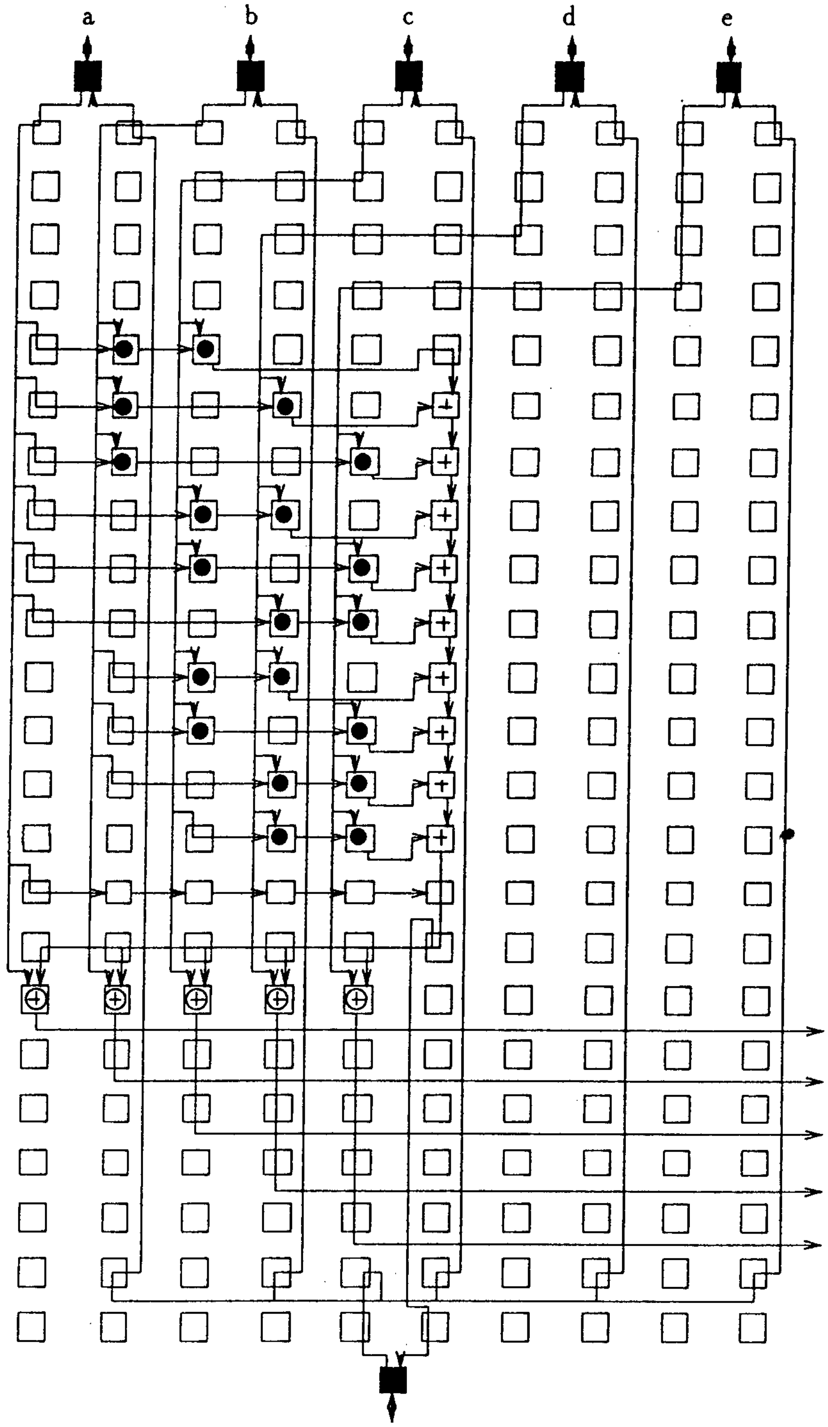


Figure 7.12: FPGA Core Design

put is routed from $cell(14,6)$ to the bottom I/O. The outputs of $cell(17,1)$ to $cell(17,5)$ are the disagreement signals; these signals are shown being routed to the right where each signal is ORed with the remaining respective error signals from its module output bits to create a single error signal for each module. This results in an N-bit vector.

For tasks in the FT mode, the FPGA inspects the N-bit vector and indicates when no majority exists (NMR function) or a miscompare (duplication with comparison function) occurs. The FPGA does this by feeding the vector to an N-bit summer whose output vector, SUM, indicates the number of errors. Clearly, a FT task using n $(n \leq N)$ redundant CMs must satisfy the condition $SUM < \lceil n/2 \rceil$ before its output can be accepted. Otherwise, the fault tolerance capability for the task has been exceeded. The FPGA performs the comparison described above, and raises the failure signal if the condition is not met.

In Subsection 7.2.1, we described the conversion of a 5MR voter to a 3MR voter by assuming we could force gates to output constant 1s or 0s. When the gates are implemented using FPGA cells, their outputs can be programmed to output constant 1s or 0s.

The reader can verify that the FPGA output is the majority of b, c, and d and that the proper error detections are enabled when the following steps are performed:

1. Program $cell(1,1)$ to output a constant 1 and $cell(1,9)$ to output a constant 0.

2. Program $cell(17,1)$ and $cell(17,5)$ to output a constant 0.

Now, let us assume that modules a and e are being used in the duplication with comparison mode. To switch from 3MR operation to duplication with comparison using a and e, the FPGA cells in the 3MR configuration that are different from the core design are first programmed back to their original state. To create the compare function for a and e, the following steps are performed:

1. Program $cell(15,5)$ to route e from the bus to its output.

2. Program $cell(17,2)$, $cell(17,3)$, $cell(17,4)$, and $cell(17,5)$ to output a constant 0 to disable all error detection except between a and e.

This example demonstrates how the core design of the FPGA can support, directly in hardware, fault-tolerant operations on all N modules and every combination of fewer modules.

The FPGA is also configured to recognize the presence of a 0 *message_destination_bit.* Programming $cell(14,6)$ to output the opposite binary value from the one to be recognized causes the XOR cell to raise the detection signal when the desired bit is found. By making the *message_destination_bit* the lower-order bit in the mailbox, the higher-order bits in the message can be ignored during polling so that the cell corresponding to $cell(14,6)$ outputs a constant 1 for detection of a 0 *message_destination_bit.*

When the MC issues a write/read to any RM, the bidirectional I/O pins in the FPGA are switched accordingly. When the data source for a write operation is either the host or the MC, routing in the FPGA is from the bottom I/O pins to the top I/O pins of Figure 7.12. Figure 7.12 is a representative layout; a more compact routing of signals through the FPGA to the RMs exists but the layout shown here is easier to follow.

To show how the FPGA supports the transfer of data between RMs for message passing we refer to the collection of cells near the bottom I/O pad of Figure 7.12. The I/O pad and six cells corresponding to the bottom of Figure 7.12 are enlarged in Figure 7.13. This figure shows that the output of the FPGA can be stored by using a cell as a flip-flop. Another cell serves as a two-input multiplexer with output select provided by the MC. Using the circuit of Figure 7.13, data read from the source RM into the flip-flop can be routed through the multiplexer to the FPGA/RM interface. From there, the data are written to the destination RM(s). Note, that the data could also pass through the voter for fault-tolerant message passing within a combined FT and MP application. Once intertask communication is initiated, the circuit of Figure 7.13 allows minimum movement of data for message transfer — from source RM, to FPGA, to destination RM.

7.2.3 FPGA Support for Software Fault Tolerance

We have demonstrated how FPGA cells can be programmed and reprogrammed to provide a virtual FPGA that is much larger than the physical FPGA. In the context of dependable computing, our FPGA-based approach shows promise of significant performance gains over traditional software-intensive approaches. We apply this capability to the enhancement of software fault tolerance.

Software components are especially susceptible to unanticipated faults. For example, the Apollo lunar missions were one of the most carefully planned and executed software projects ever undertaken; yet nearly all the major problems in the series were

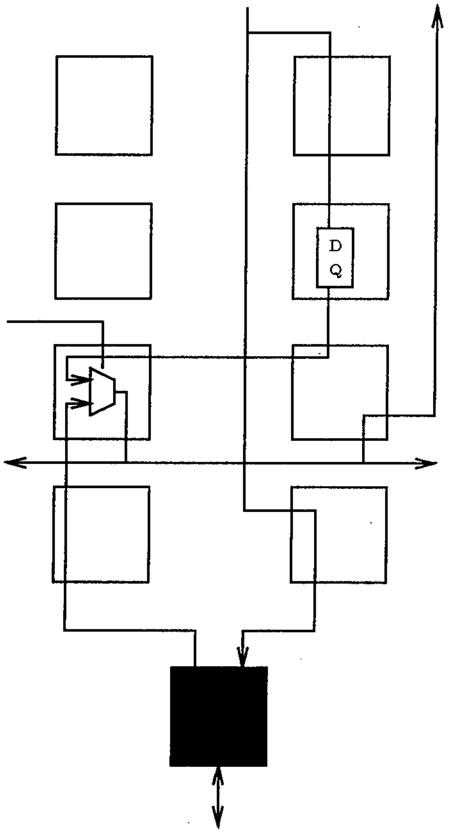


Figure 7.13: Circuit for Intertask Message Transfer

attributed to software design faults [59]. Myers [60] also points out that the United States' first space mission to Venus failed because of a missing comma in a FORTRAN Do statement. Tolerating faults of this type, through software fault tolerance, requires system adaptability.

A highly configurable digital system adapts to various computational tasks through reuse of its available hardware for maximum performance with minimum overhead. Adaptive systems offer attractive features such as conservation of board space and the migration of algorithms from software to hardware. These systems can exploit reprogrammable FPGA technology by continually reconfiguring the existing hardware, and thus reduce the amount of hardware dedicated to system functions.

A major obstacle for the designers of adaptive systems to overcome is that current FPGAs are still too small to be useful when used singly. Another is that most FPGAs require complete reloading of the program memory when changing even a single programming bit, requiring them to be offline while they are programmed. To permit systems to become more adaptive, FPGA manufacturers are striving to produce devices that are more versatile. As mentioned earlier, Atmel has an FPGA product line that supports dynamic reconfiguration. Recently, Xilinx also announced an FPGA family that has on-the-fly programming capability [61].

Next, we describe how FPGA dynamic reconfigurability is used to satisfy complex checking required for software fault tolerance. Through use of our simulation tools, FPGA dynamic reconfiguration is verified.

7.2.4 Applying Dynamic Reconfiguration

The underlying cause of failure of a software component is in its design [22]. Software fault tolerance therefore encompasses various techniques to mitigate the fault; in each of these techniques extra software is added to the component to make it fault-tolerant. A critical part of this software checks if the component has experienced an error so that initiating a corrective action allows fault tolerance to take place. Because the faults being tolerated are design faults, the checking routine is intrinsic to the software component, and the routines are as varied as the software components themselves. We are not discussing the notion of software fault tolerance *per se*, but are using it only as a demonstration vehicle for our proposed architecture. We have considered the case of providing hardware fault tolerance in a flexible manner in Section 7.2. A wide variety of functions are necessary to implement software fault tolerance, yet a software-intensive solution is not the only choice simply because of the large number of functions involved.

Here, we show how a single dynamically reconfigurable FPGA can provide the same functionality as a software-intensive solution, but the checks are performed with the speed of a hardware solution. The components of Figure 7.10 form a coprocessor to a processor that executes the application software. Checks of the application software are done in the coprocessor. Separation of the application and its associated fault-detection mechanisms off-loads the checking overhead from the executive processor to its coprocessor. This allows the executive to proceed while the coprocessor checks for an error. In the rare event when an error occurs, the coprocessor notifies the processor to halt and rollback to the point in its execution when the error was detected. Independent units can also be used to reduce the number of failures that can defeat fault-tolerant strategies where the processor has the responsibility of checking its own execution [25].

In this section, the two main software fault tolerance techniques, *N-Version Programming (NVP)* and the *Recovery Block (RB)* method [22], are considered and examples of the types of checking routines used by each are described.

7.2.5 N-Version Programming

NVP consists of N functionally-identical versions of a program ($N > 1$) which have been independently designed to satisfy a common specification. The N versions are executed and their results compared by some form of replication check [22]. In NVP, this check is often referred to as *inexact voting* because the correct outputs of the N versions may not be exactly the same, but are nevertheless all considered to be correct due to tolerances in the specification. An algorithm for inexact voting could be very complicated, and a general algorithm for determining if the N inputs are different from one another within an allowable range that is applicable to any application is not possible to formulate [62].

A function that checks that an output is within a certain range is an elementary form of inexact voting [63]. Figure 7.14 shows the block diagram for the checking circuit where, in addition to the input data being checked, the LB input specifies the lower bound, and the UB input specifies upper bound.

Figure 7.15 shows how the checker for either the upper or lower bound is decomposed into a bit-slice for each bit of an n-bit input vector.

In conventional cell-based VLSI design, a macro with the gate-level representation

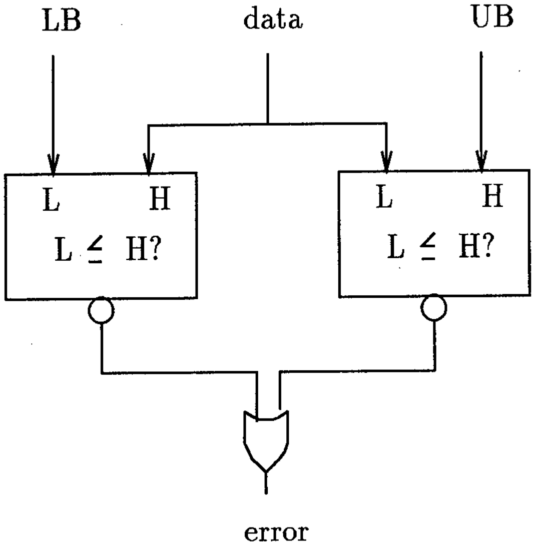


Figure 7.14: Bounds Checking

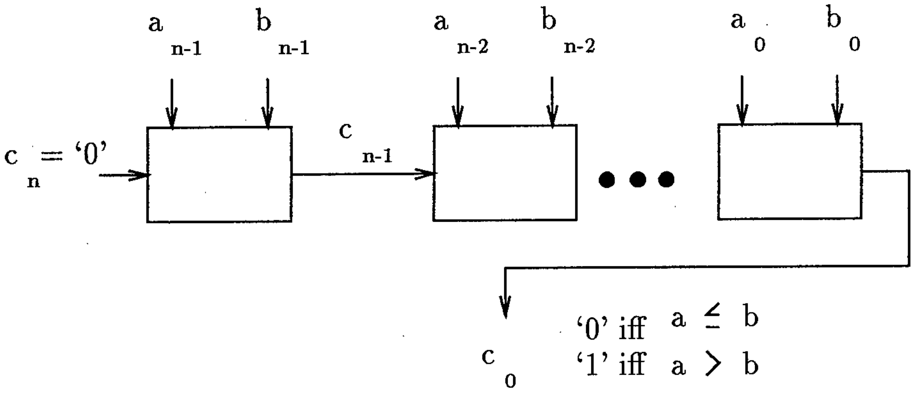


Figure 7.15: Bit-Slice Design for Bounds Checking

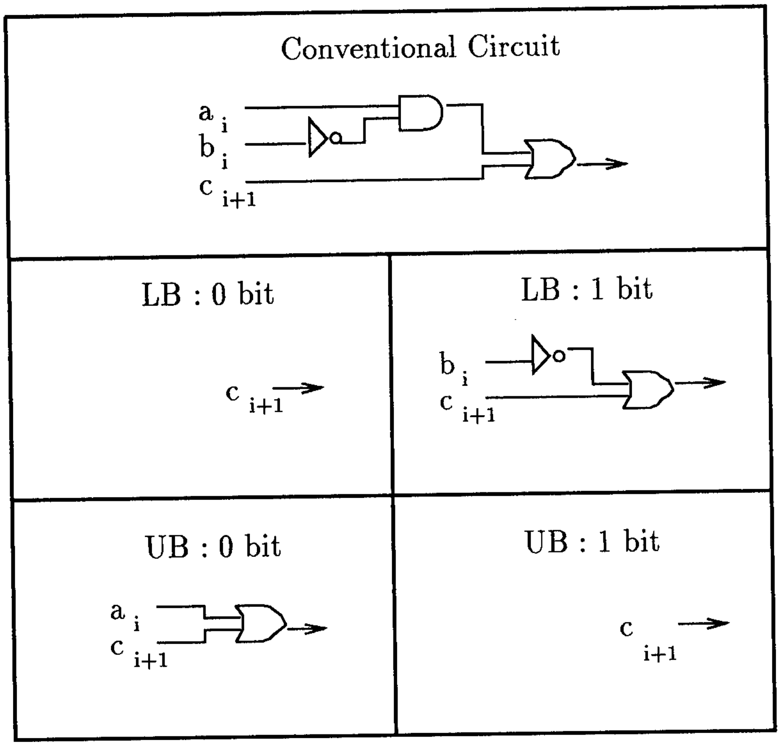


Figure 7.16: Data-Folding

shown at the top of Figure 7.16 would be used for each stage. In a custom design, the penalty for routing the inputs for data, *LB* and *UB* to the checker is much less than in a direct FPGA implementation. More circuitry is needed in the FPGA because the circuit's layout has to be spread over a wider area in the array to allow sufficient access by the programmable interconnect. Also, several FPGA cells are needed for routing so the combined effect is a circuit that is less efficient in chip area and has much greater path delay. However, the partial-reconfigurability property of the FPGA raises some interesting possibilities for a more economical realization.

A scheme called *data folding* [64] allows a circuit that performs a function on specific data to be optimized for those data; that is, constants can be "hardwired" into the circuit. In the case of the range check, the LB and UB are constant for the set of input data so that an alternative FPGA implementation offers better routing opportunities. Figure 7.16 shows how the stages are modified based on each bit value of the bounds. Once the bounds are folded-in, at most two signals are needed per stage and the number of gates that the FPGA has to implement per stage is also at most two. In the FPGA implementation, the checking operation must stop when the bounds change to give time for the FPGA to partially reconfigure. The time to reconfigure depends on the Hamming distance between the new and old bounds. The following three subsections[2] of Section 7.2 illustrate reconfiguration occurring concurrently with device operation — we refer to this capability as *configuration parallelism.*

7.2.6 Recovery Block (RB)

In this technique, different algorithms, called *alternates*, are available to perform a given task. Associated with these algorithms is an acceptance test, and together they comprise a recovery block. When an RB is entered, the primary alternate executes first, and, if its results are not acceptable, then the alternates are sequentially invoked until the acceptance test is passed. The acceptance test is the most critical component of the RB and it must be complete enough to evaluate the performance of each of the alternates [62].

The following example of an RB has been frequently used [22] [62] to demonstrate

[2] ©1996 IEEE. Reprinted, with permission, from *Proceedings of the Sixth Great Lakes Symposium on VLSI,* Iowa State University, Iowa, March 22-23, 1996, pp. 39-42.

this technique:

```
ensure
    SortCheck (S) and CheckSum (S, Prior(S))
by quickersort (S)
else by quicksort (S)
else by bubblesort (S)
else error
```

This RB is for performing a sort on a list, S, of elements. The acceptance test performs two checks: are the data sorted correctly, and are the sums of the elements from the sorted and unsorted list the same? The function *SortCheck* verifies the order of the elements in the sorted list. The function *CheckSum* sums the elements of S and Prior (S) and checks for equality. This ascertains that no elements were dropped or inadvertently repeated during formation of the sorted list.

We now propose an FPGA implementation of the checking functions. Figure 7.17 shows the block diagram of the circuit to be implemented. Note that *CheckSum* requires the data prior to *and* after the sort, while *SortCheck* needs the data only after the sort is complete. This functional latency allows us to exploit configuration parallelism: *SortCheck* is configured into the FPGA while the device performs *CheckSum.*

Figure 7.18 shows a stage of a bit-slice design for the checking functions. The top three rows of the array perform *SortCheck*, and the remaining seven rows perform *CheckSum.* As shown in the the block diagram of Figure 7.17 it is necessary to configure a cell at the output stage of *SortCheck* as a latch flip-flop in order to capture the error signal.

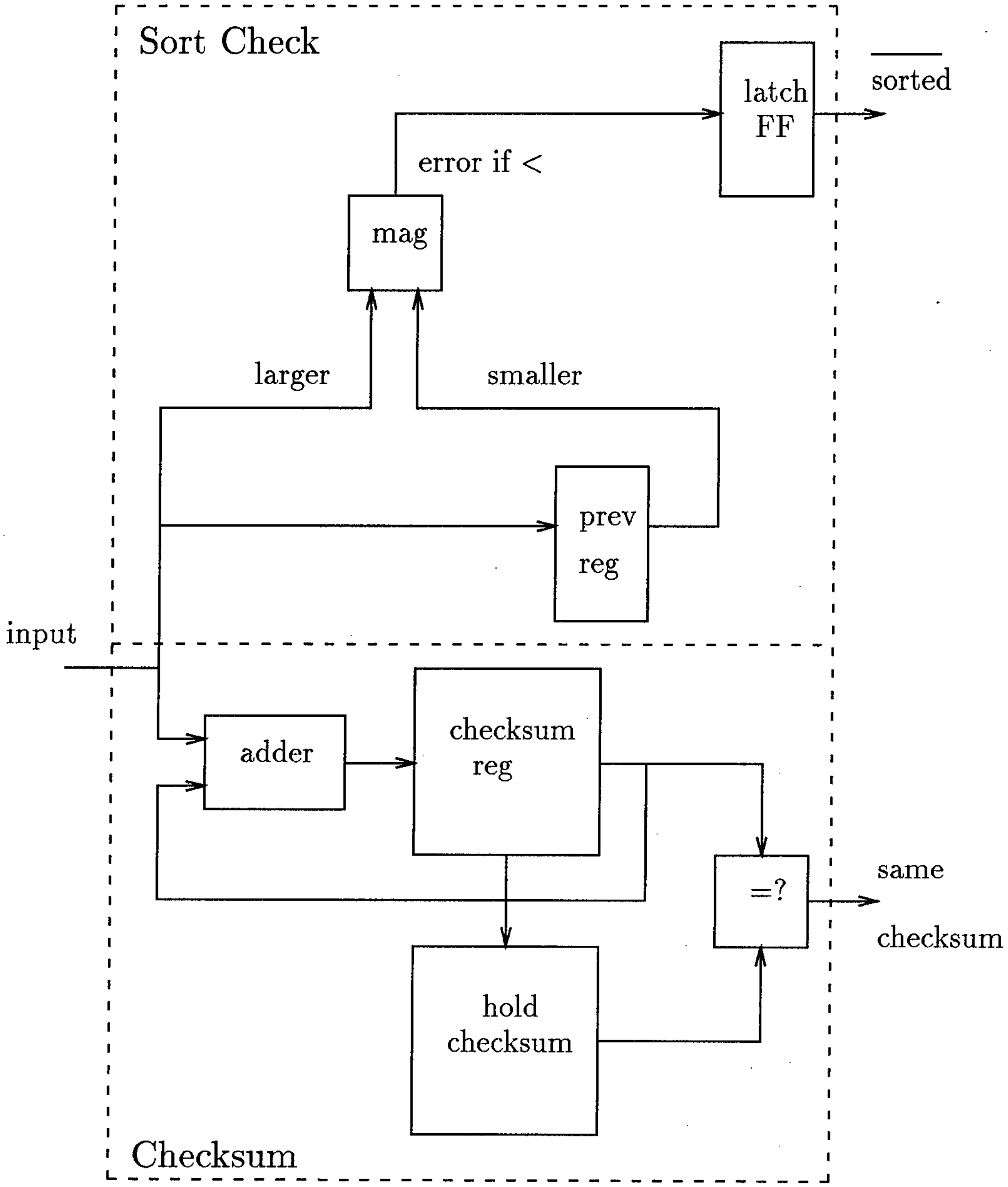


Figure 7.17: Block Diagram for Sort Check and Checksum

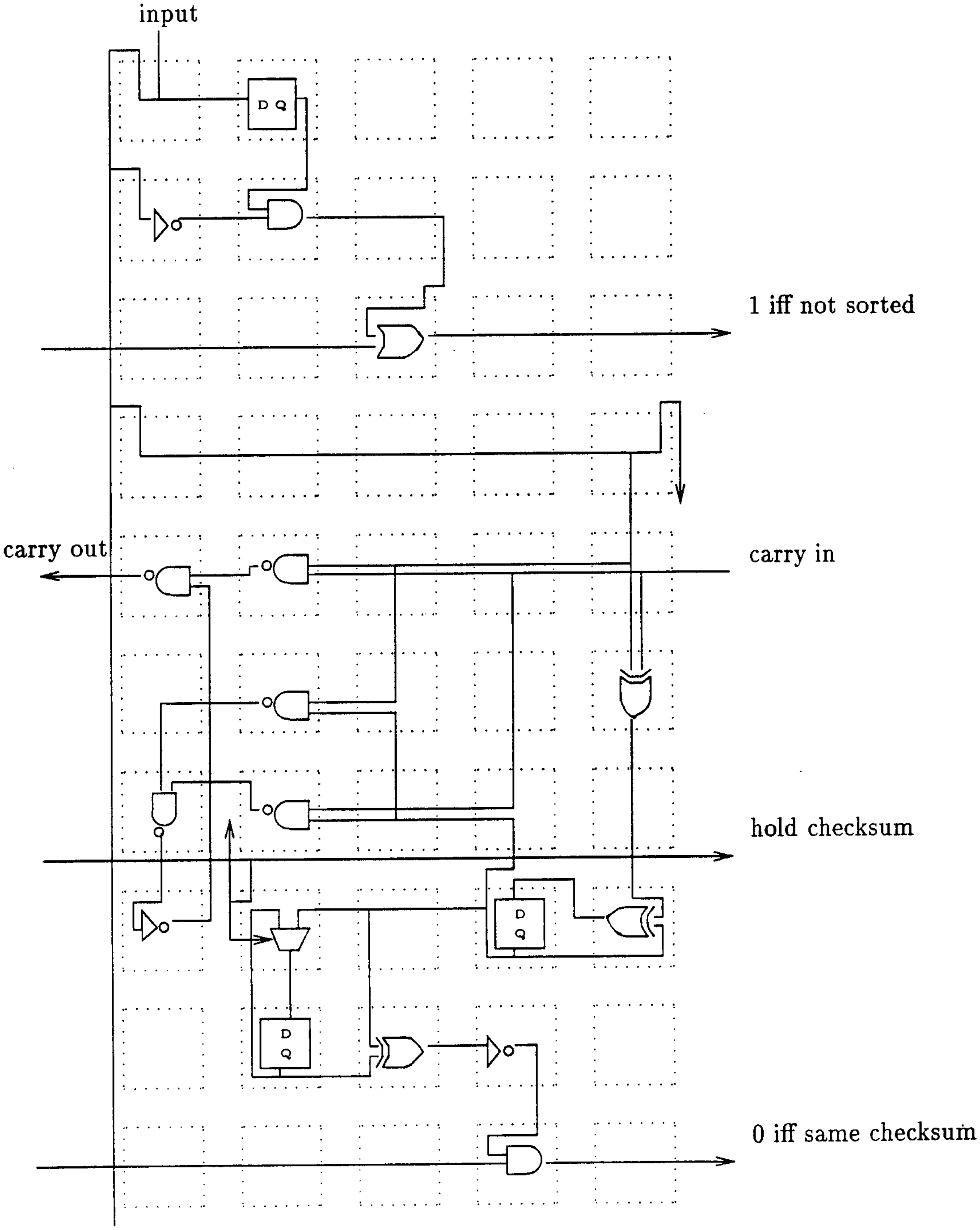


Figure 7.18: FPGA layout for Sort Check and Checksum

Referring back to Figure 7.9, a simplistic approach to configuration parallelism is to partition the chip into distinct areas as shown in the figure. This is the same as creating a chip within a chip and is essentially no different than having two non-dynamically reconfigurable FPGAs that are programmed independently. In the more versatile approach for FPGA programming, the functions are not necessarily confined to distinct chip partitions; instead the areas that they occupy may overlap. In these instances, configuration parallelism stops once either the row or column coordinate of a cell to be programmed intersects with the area of the active function. Programming is resumed when the active function has completed. For true configuration parallelism, any available logic resources should be of potential use; however careful preplanning is required. For example, the cell in the upper-left corner of Figure 7.18 is the fanout origin of the input signal. Though this cell is in the area of *SortCheck*, it is necessary to configure it prior to initiating *CheckSum.* Configuration of the remaining cells for *SortCheck* take place during operation of the *CheckSum.*

Dynamic reconfiguration, especially configuration parallelism, is complex and simulation tools to support it are indispensable. For instance, if a cell of one function is driving a bus while another function is being configured in parallel, then no cell of the new function that will drive the bus can be configured until the current function is complete — else a bus conflict will occur. The VHDL simulations reveal when such conflicts occur by forcing an unknown value ("X") on the bus. Attempting to debug such a design at the actual hardware level would potentially result in an unreliable design and can cause damage to the FPGA.

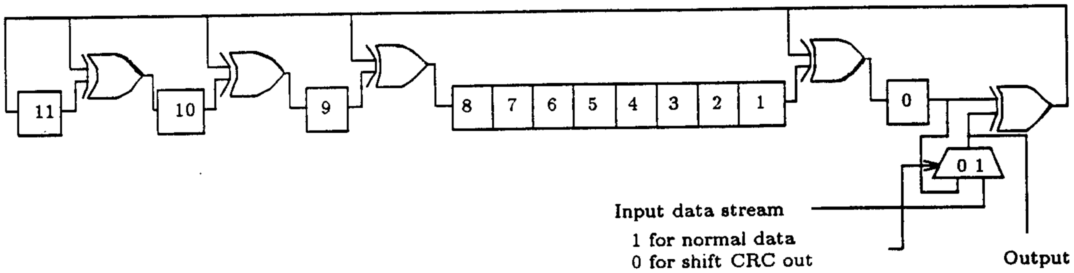


Figure 7.19: Circuit for Cyclic Redundancy Check (CRC)

7.2.7 Preserving Configurations

As a final example of ways dynamic reconfiguration can take place, we consider an extension to the example of Subsection 7.2.6. Assuming that the RB has been performed successfully, we now wish to provide reliable transmission of the sorted data. We do so by having the FPGA perform a cyclic redundancy check (CRC). The encoding for the CRC is performed independently and in parallel over the several serial-bit streams. The CRC check bits for each bit stream are generated during transmission and are appended to the end of the block [65]. A 12-bit linear-feedback shift register encoder for a CRC-12 cyclic code is shown in Figure 7.19.

The FPGA implementation is shown in Figure 7.20. By overlaying this layout onto the layout of *CheckSum* shown in Figure 7.18 we see that not all of *CheckSum*'s cells need changing. In fact, 18% of the cells in *CheckSum* have the same functionality in both designs. Preserving cell configurations in *CheckSum*, by programming only the differences, reduces the time to configure the CRC-12 encoder by 25%.

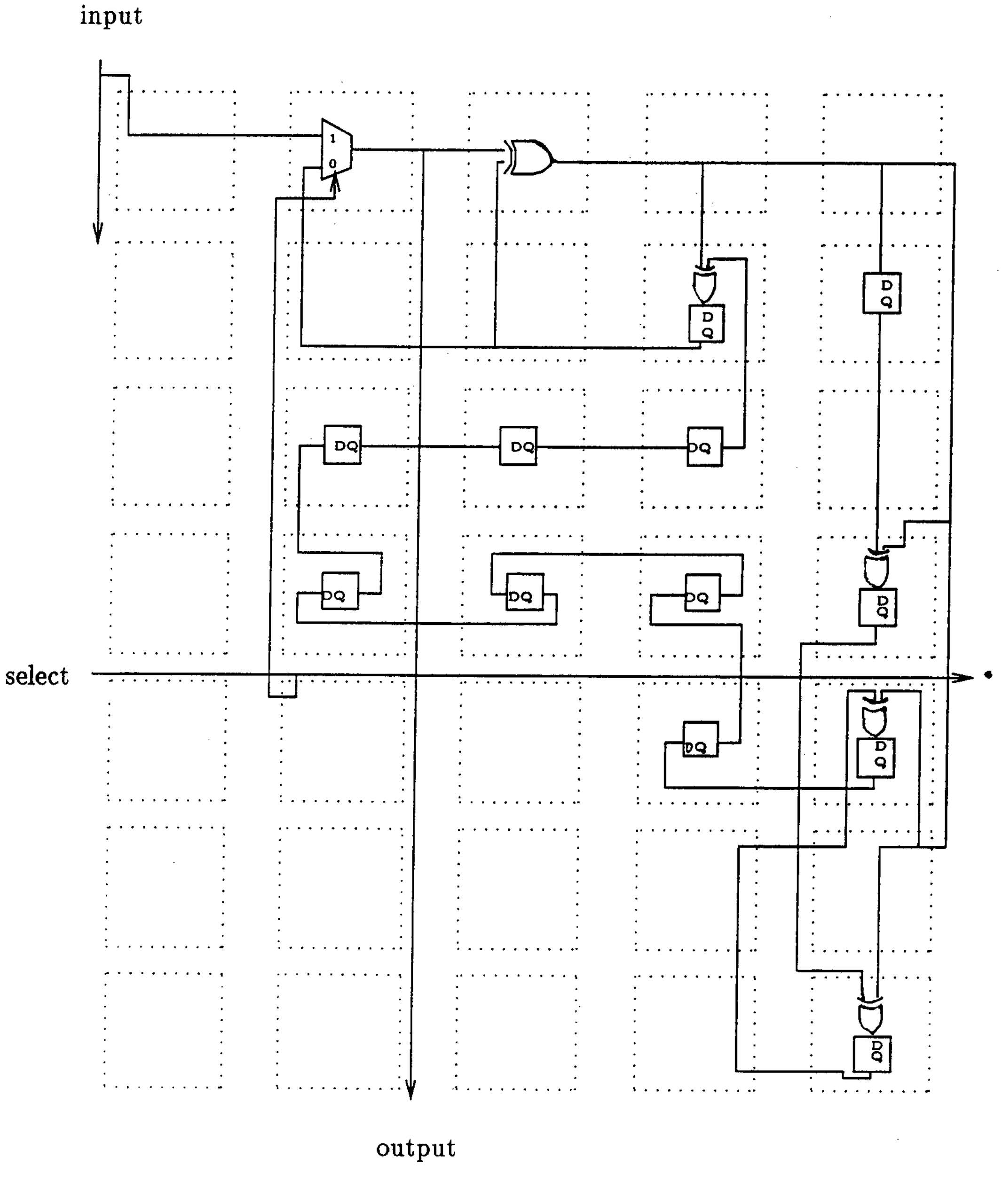


Figure 7.20: FPGA layout for CRC

7.2.8 Performance

In addition to using logic simulation to verify the use of dynamic reconfiguration, simulations also show the performance payoff that can be expected. While it can be generally assumed that the performance of a hardware-based solution will outperform that of a software-based approach, the reconfiguration time for the FPGA is still significant and must be considered. There are delays associated with partial reconfiguration, and also with parallel configuration. When the length of the data stream is insufficient to allow a new function to be completely configured, then processing stops when the new function is needed and resumes when configuration is complete.

As shown in this section, the FPGA can be configured for many different functions, and dynamic reconfiguration can take several forms: partial reconfiguration with data folding, configuration concurrently with device operation, and partial reconfiguration by overlaying layouts. To determine the suitability of using the FPGA, we use the VHDL model shown in Figure 7.10 and simulate the stochastic properties needed to realize numerous job mixes. Execution statistics of each job are determined. The controller sequences the FPGA through the necessary reconfigurations that support the particular job mix. The output data produced by each job are input to the FPGA-based system for checking. One characteristic of a job is the amount of data it produces; by varying the amount of output data produced by each job we can examine the combined impact that data stream length and dynamic reconfiguration have on performance.

Figure 7.21 shows the time required by each job mix to have its output data checked as a function of the amount of output data produced by the mix. Equivalent software versions of the FPGA functions were written in an assembly language [42] and executed to verify their correctness. Based on the instruction counts, the corresponding completion times for processors with specified MIPS (Millions of Instructions Per Second) ratings are also shown in Figure 7.21.

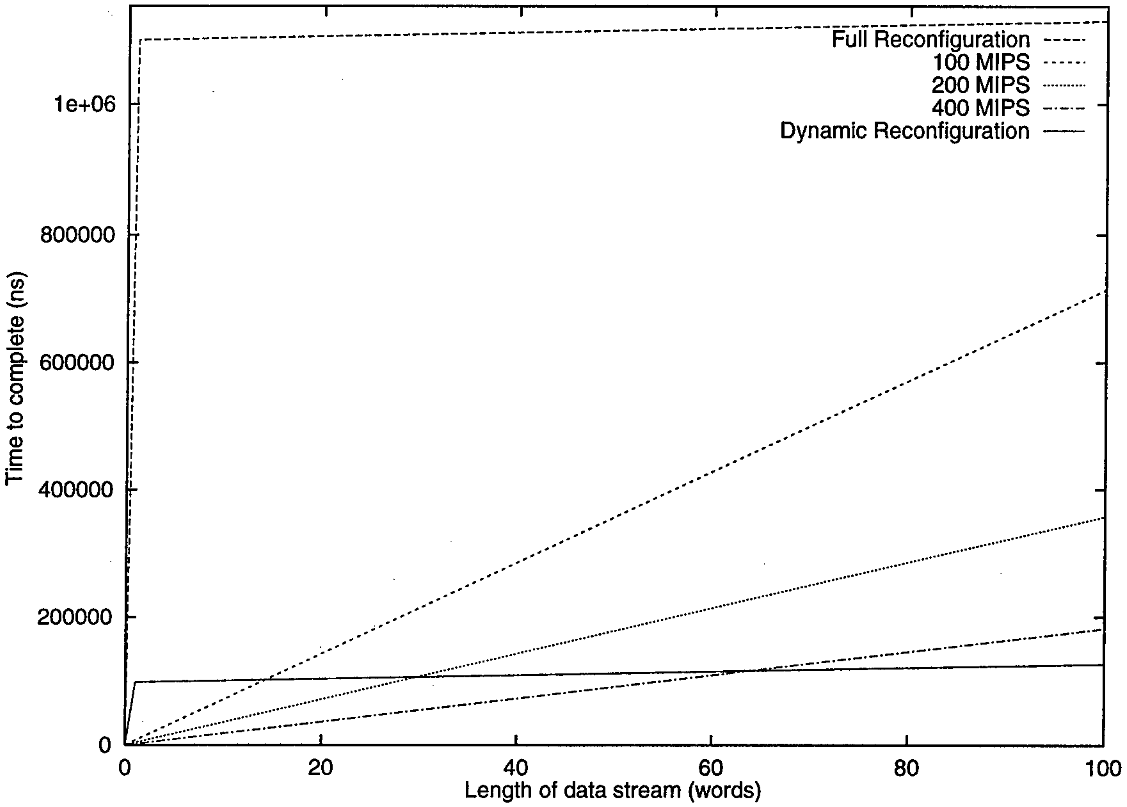


Figure 7.21: Data Stream Length vs. Completion Time

This figure establishes that, as long as the data stream length per job exceeds a certain length, then for practical purposes, the FPGA's throughput overtakes that of a processor executing at any stated MIPS. Furthermore, we observe that these crossover points occur at relatively small values of the data stream length.

Also shown in Figure 7.21 is the time required for a non-dynamic version of the

FPGA to complete servicing the job mix. In this case, all of the used cells must be reprogrammed when switching between the different configurations even though only a part of the array needs to be changed. Comparing completion times of an FPGA to software, the crossover points in Figure 7.21 occur at much smaller values of data stream length for the FPGA that is dynamically reconfigurable than for the one that is not. Dynamic reconfigurability presents more opportunities for an FPGA to be used as an application accelerator.

7.2.9 Summary

In this section we have demonstrated ways that a dynamically reconfigurable FPGA can accelerate software algorithms. The checking algorithms found in software fault tolerance techniques were used as examples; their variety indicates that the FPGA can provide exactly the specialized machines needed for a particular application. These machines execute with the inherent parallelism of hardware while avoiding the overhead associated with load/store, branch operations, and instruction fetch and decode.

Recently [66], it was noted that appropriate architectures need to be developed for reconfigurable computing. Using the simulation tools and the types of experimental results mentioned in this section, we have developed dynamically reconfigurable architectural support for selective fault-tolerant and parallel computing systems.

7.3 Fault Simulation

Although Field-Programmable Gate Arrays (FPGAs) are tested by their manufacturers prior to shipment, they are still susceptible to failures in the field. In this section, test vectors generated for the emulated (i.e., mission) circuit are fault simulated on two different models: the original view of the circuit, and the design as it is mapped to the FPGA's logic cells. Faults in the cells and in the programming logic are considered. Experiments show that this commonly-used approach fails to detect most of the faults in the FPGA.

7.3.1 FPGA Considerations

Field-Programmable Gate Arrays (FPGAs) resemble traditional mask-programmed gate arrays, but differ in that they are programmed by the end user.

The starting point for designing an FPGA is logic entry. A schematic capture or logic synthesis tool is used to create a description of the circuit to be implemented. Then, the circuit design is translated into a standard form consisting of basic logic gates. This process of converting the netlist of basic logic gates into a netlist of FPGA cells is referred to as *technology mapping* [67].

For this section, we define the following terms:

unmapped logic circuit A circuit design described as a netlist of basic logic gates. This is the target circuit that is to be emulated by the FPGA.

mapped logic circuit The same logic design as an unmapped logic circuit, but implemented as FPGA cells through technology mapping.

mission vectors The vectors that would be applied to the unmapped logic circuit. These vectors are applied to the corresponding inputs of a mapped logic circuit after the FPGA has been programmed. In the case where both circuits are fault-free, their output responses for any sequence of mission vectors are the same.

Manufacturers' documentation of reprogrammable FPGAs state that the devices have been tested with 100% fault coverage prior to their shipment (e.g., [49] [50] [51]). However, faults induced thereafter (i.e., field failures), must be tested for by the user. The manufacturer's test algorithm may be unavailable, but even if it were available, it may be impossible to apply it during board level test due to the FPGA's pin assignments. Thus, the most common approach is to generate test vectors for the unmapped circuit that the FPGA will embody. This section describes the adverse effects that technology mapping has on this approach.

7.3.2 Determining Fault Detection Coverage

The procedure we use to measure the effects of technology mapping on fault detection coverage in a reprogrammable FPGA is as follows:

- *Step 1:* Create a logic model of the unmapped circuit.
- *Step 2:* Perform the technology mapping.
- *Step 3:* Create a logic model of the unprogrammed FPGA consisting of cells, cell interconnections, cell program memory, and FPGA programming logic.

- *Step 4:* Create the programming vectors that embed the mapped circuit into the FPGA model.

- *Step 5:* Obtain mission vectors that achieve the maximum fault coverage for the unmapped circuit, and produce lists of detected and undetected faults for the unmapped circuit.

- *Step 6:* Using vectors obtained from *Step 4* and *Step 5*, obtain FPGA simulation vectors.

- *Step 7:* Fault simulate the vectors from *Step 6* on the mapped circuit, and obtain lists of detected and undetected faults in the FPGA.

- *Step 8:* From the lists of detected and undetected faults for both circuit types, determine the discrepancies in fault coverage.

Next, an illustrative example of a full adder mapped to an FPGA provides a demonstration of this procedure.

7.3.3 Circuit Mapping for Fault Simulation

The first step in our procedure involves determining the logic diagram of the circuit under study. Figure 7.22 shows a full adder composed of basic logic gates. This represents how the designer might describe the circuit to be implemented in the FPGA. From this schematic, the logic model of the unmapped full adder is created.

Technology mapping is the next step in measuring the fault coverage detection. Figure 7.23 shows a mapping of the full adder description to the FPGA where the

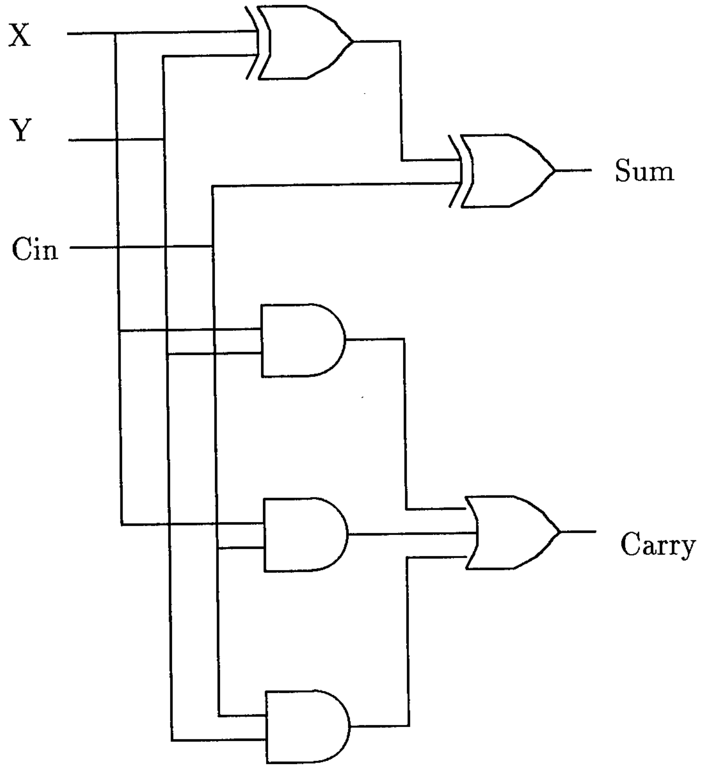


Figure 7.22: Unmapped Full Adder

set of logic gates specified in the original schematic have been transformed to those available through cell programming. While placing the cells that implement the full adder, additional cells are allocated for signal routing. Cells that are not involved in the full adder implementation have their bus drivers disabled.

In Figure 7.23, individual cells are identified by their row and column coordinates, indexed from 1, with *cell (1,1)* being the cell in the top left corner.

7.3.4 Simulation Setup

Logic models of the unmapped and mapped circuits are needed prior to fault simulation. The NIF language (see Section 7.1.4) was used to model both circuit types. A computer program was written to generate automatically the NIF description of the unprogrammed FPGA. A gate-level NIF model of the unmapped full adder was created and then translated to the Navy's Hierarchical Integrated Test Simulator (HITS) [58] language for fault simulation. The simulation platform was a VAX 8650.

To reduce both the model complexity and the simulation times, the unused cells were modified. An unused cell can be greatly simplified because the only path for its outputs to the adder output is through the bus interface, which is disabled. As a result, a cell that was not used by the technology mapper and router was replaced by a *stub* of a full cell. All programming logic was removed from a stub and its architecture was reduced to only disabled bus drivers and a single gate that sinks all the cell's inputs and sources a constant output of zero. For the full adder the elapsed time for fault simulation of the mapped circuit (2,312 gates) exceeded 18 hours.

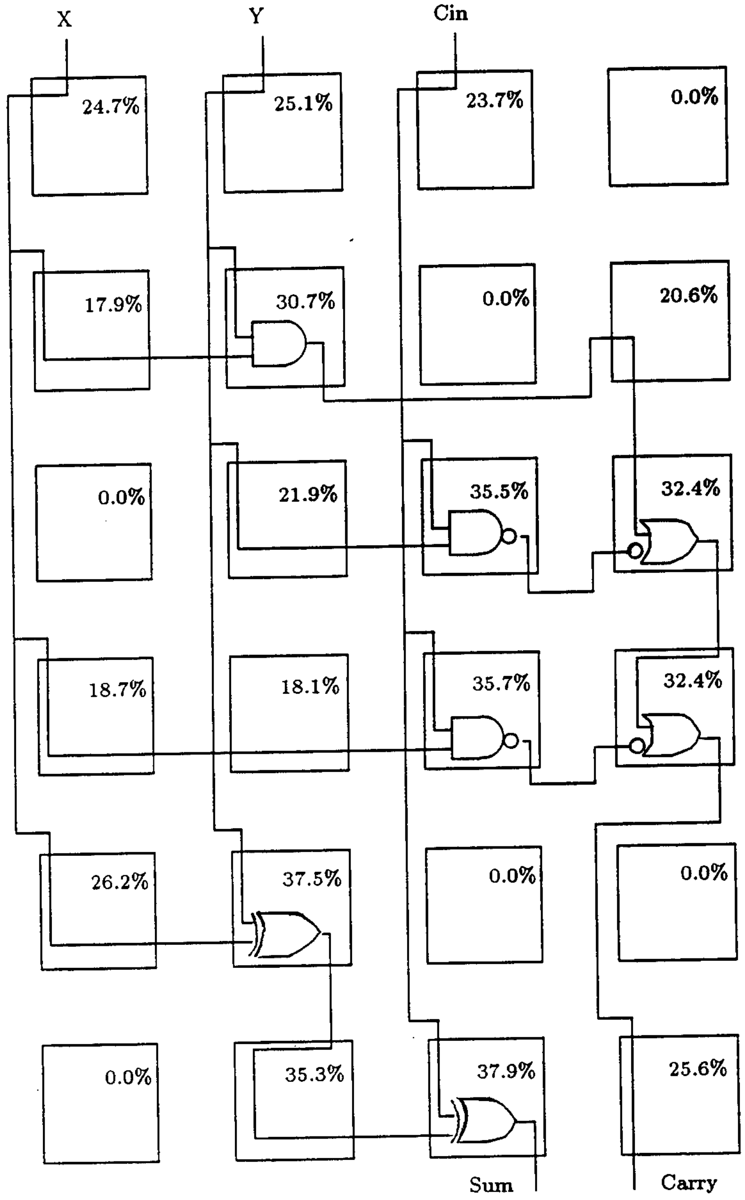


Figure 7.23: Mapped Full Adder. Percentages are individual cell fault coverages achieved by applying an exhaustive set of mission test vectors.

An exhaustive set of mission vectors achieved 100% fault detection of all single stuck-at-0 and stuck-at-1 faults in the unmapped circuit; this set comprised the mission vectors for our experiment. A vector sequence that programs the FPGA to implement the full adder was created (882 vectors), and the eight mission vectors were appended to this sequence. This sequence of 890 vectors was then fault simulated, and lists of detected and undetected faults in the FPGA were obtained.

7.3.5 Simulation Results

Fault grading was done in accordance with the standard procedure for fault coverage reporting (MIL-STD-883 Procedure 5012) [68] [69]. This procedure provides a consistent means of reporting fault coverage, regardless of the logic and fault simulator used. Exceptions to the baseline procedure were as follows: the fault universe was based on all faults on the signal lines, instead of fault equivalence classes of those faults; undetectable faults were not dropped from the fault universe; and faults in the logic that fed only the bus enables (there were 95 such faults in each cell) which are detectable only as potential detects were dropped from the fault universe.

The final step of the procedure calls for determining the fault coverage discrepancies between the two circuit types. Table 7.1 shows the fault coverages on a cell-by-cell basis, and for the programming logic; these values are repeated in Figure 7.23 where the corresponding function of each cell is also shown.

Unfortunately, when the constant zeroes sourced by the stubs are propagated through the logic model, these constants cause the fault simulator to reduce the fault universe, and as a result the number of faults per cell is not constant. Likewise, cells on the FPGA's periphery have some of their inputs tied to a constant value, and this accounts for the differences in the total number of simulatable faults for these cells. After adjusting the fault universe to account for these artifacts of fault simulation, only 17.5% of the faults in the entire FPGA were detected.

No FPGA faults were detected until the first mission vector was applied. Most faults in the programming logic are detectable only as potential detects. Unprogrammed cells that are intended to be programmed produce indeterminate values in the simulated circuit, so the fault originating the error is only potentially detectable. However, if a programming logic fault results in unmapped and mapped circuits that are decisively not functionally equivalent, then the fault is still detectable as a solid detect.

The technology mapping illustrated by Figure 7.23 is not unique. It is well-known that layout can influence testability (e.g., [70]), and a different arrangement of the stubs would produce different fault coverages for their neighboring cells. The *technology mapping* determines what programming input a cell can receive in the presence of a fault, so another technology mapping alters the circumstances of detecting the fault. Furthermore, a change in the *order* in which cells are programmed can also produce differences in fault detection.

The fault simulation results show that, when a set of mission vectors that is a complete test for an unmapped logic circuit is applied to a mapped circuit, the fault coverage is greatly reduced. To determine if the low fault coverage for the mapped circuit was due to intrinsic lack of testability of the FPGA, an experiment was performed to determine the achievable level of fault coverage for an FPGA cell. An FPGA cell (without the bus drivers) was modelled and a test of size 12 was obtained that detects all detectable faults in the combinational part of the cell. Next, this test set was modified so that whenever a vector is applied to the cell, the cell's flip-flop is clocked, resulting in 24 vectors. A vector was then added to test the flip-flop reset, making the sequence length 25. In the worst case, each of these 25 tests would require a unique cell programming. A total of 25 clock cycles (24 to shift in the data and one additional cycle to load the hold register), multiplied by the number of vectors, means that at most 625 test vectors would be required to detect all detectable faults in a single FPGA cell. Using this test sequence, the maximum achievable fault coverage for a cell is 95.884%. This coverage represents detection of faults in both the cell architecture and the cell's program memory. A cell is therefore intrinsically highly testable; however, the low fault coverage achievable using a complete test vector set for an unmapped circuit demonstrates the scope of the adverse effect technology mapping has on fault coverage.

7.3.6 Summary

The effectiveness of user tests applied to an FPGA depends on the cell architecture, the cell program memory, and the FPGA's programming logic. The approach shown in this section can be applied to study other reprogrammable FPGA architectures. We have demonstrated that, when a gate-level design is mapped to a network of FPGA cells and tested using mission vectors developed for the original, unmapped gate-level design, the reduction in fault detection coverage is enormous.

7.4 Reliability Analysis

It is well known that the reliability of a fault-tolerant system cannot exceed the reliability of the output stage of the system. Put another way, the reliability of the element that resolves the outputs of the redundant CMs to produce the system output governs the overall system reliability. The design that we have targeted offers multiprocessing and flexible fault tolerance. Previous analyses showed that the FPGA is superior to a software-intensive, dedicated processor. This analysis shows the FPGA to also offer superior reliability.

We use a widely accepted reliability-prediction method [71] to calculate the failure rate of the output stage of DRAFT.

The failure rate, λ, is calculated by the formula:

$\lambda = (C_1\pi_T + C_2\pi_T)\pi_Q\pi_L$ Failures/10^6 Hours

Where:

1. C_1 is the die complexity failure rate.
2. π_T is the temperature factor.
3. C_2 is the package complexity failure rate.
4. π_E is the environmental factor.
5. π_Q is the quality factor.
6. π_L is the learning factor.

In the calculations that follow, the following values are constant: $\pi_T = 0.16$, $\pi_E = 0.5$, $\pi_Q = 3$, and $\pi_L = 1$.

We calculate the failure rate for the two cases: the reconfigurable FPGA approach and a conventional approach with fixed hardware logic. A tradeoff occurs by migrating complexity from gates in the case of a fixed hardware design to bits in an FPGA design that is continuously reconfigured so that its *gates* are reused over and over again. For the fixed hardware design, C_1 is measured by the gate count. Only when external storage is added can the FPGA implement usable *gates*; so initially C_1 for the unprogrammed FPGA is based solely on the FPGA's cells, program memory and program logic. By adding external memory, we increase the C_1 factor of the reconfigurable design. For external memory we turn to a ROM solution that offers the greatest density available — up to 16 Megabits per chip. For example, 1 Mbits of ROM can be used to implement 41,600 *gates*. The C_1 value for a ROM of this size is only 0.0052, while the C_1 value for the equivalent number of hardware gates is 0.29 — a factor of 55 increase in complexity.

A 1024 cell FPGA has an initial failure rate that is calculated before any *gates* are implemented. Three bytes program a cell, and it is assumed a single *gate* is implemented per cell. Increases in the gate count of the target design imply a 3 byte increase in the ROM on a per gate basis. With 16 Megabits of ROM, the FPGA can implement 666,666 *gates*. For the fixed hardware solution, two subcases are considered. The first is a single package solution and the second is a two-package solution. The failure rate calculation for the reprogrammable approach includes three packages: the FPGA, the counter, and the ROM. The number of pins for the FPGA package is 224. The counter is assigned

1,000 gates and an initial package pincount of 36. Initially, the ROM of byte size 3 (for 1 *gate*) is assigned a 16 pin package. The number of address pins for the counter and ROM are then increased with the size of the ROM needed to accommodate the gate count of the target design. Figure 7.24 shows the failure rates between a fixed hardware design and a reconfigurable design.

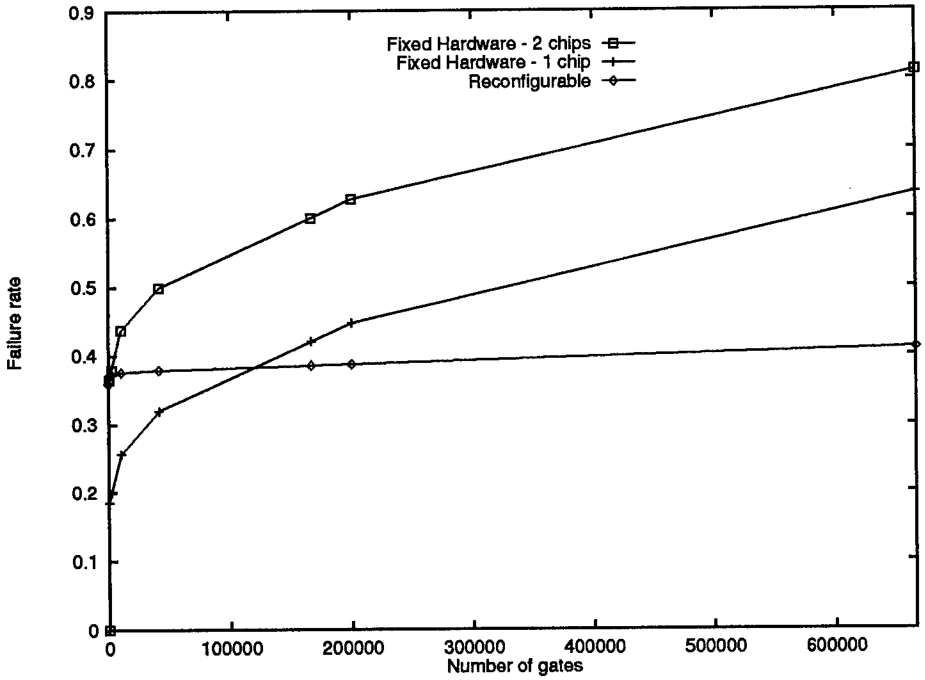


Figure 7.24: Number of Gates vs. Failure Rate

The horizontal axis is the number of logic gates required. For the fixed hardware case, this number is the same as counting the number of gates directly. For the FPGA, this is the number of *gates* implemented through dynamic reconfiguration where the unused *gates* are stored externally in the ROM. Initially, the failure rate for the unprogrammed FPGA is high due to its gate count of 55,296. However, the complexity for each implemented *gate* is placed into a significantly less complex 24-bit increment of

ROM. As a result, the failure rate curve for the FPGA, counter, and ROM grows only slightly as compared to the curves for the hardware solutions. Comparing the failure rates of the reconfigurable and fixed designs, the subcase of the single package fixed-gate solution has greater failure rates when gate counts go above 130,000. At higher gate counts, the two-package subcase would probably occur, and, as shown in Figure 7.24, its corresponding failure rate is dramatically greater than that of the reconfigurable design.

The fixed-gates case may be used to implement a processor that runs software versions of the DRAFT algorithms. Fixed gates could also implement these algorithms directly in hardware. The gate counts shown in Figure 7.24 are easily exceeded by processors with MIPs ratings in the range that has been discussed here, and the failure rates shown in Figure 7.24 do not include the memory that is required by software-based algorithms. Even so, high speed processors are often possible only when used with cache chips and memory management units in addition to the basic "CPU" functions — thus, multiple package solutions would be expected. A large number of fixed gates would be necessary for a direct hardware implementation of the algorithms. Driving up gate counts even more would be the need to match DRAFT's ability to provide an increase in the variety of fault tolerance algorithms. This is especially true when algorithm complexity exceeds that of majority voting such as when correct results may not completely agree or a reasonableness check is performed.

7.4.1 Summary

We have described a flexible architecture that supports fault tolerance and multiprocessing. A key component of the design is a reconfigurable logic and switching unit that can be reconfigured on the fly. This offers a degree of flexibility that has been achievable, up to now, only through a software implementation. Our hardware alternative has both better performance and reliability than comparable approaches. We have described how a versatile FPGA is an enabling technology for this design.

FPGA Component	Faults Detected	Total Faults	Fault Coverage
cell (1,1)	152	711	24.7%
cell (1,2)	156	717	25.1%
cell (1,3)	144	703	23.7%
cell (1,4)	0	21	0.0%
cell (2,1)	110	709	17.9%
cell (2,2)	191	717	30.7%
cell (2,3)	0	21	0.0%
cell (2,4)	125	701	20.6%
cell (3,1)	0	21	0.0%
cell (3,2)	136	717	21.9%
cell (3,3)	221	717	35.5%
cell (3,4)	201	715	32.4%
cell (4,1)	115	709	18.7%
cell (4,2)	114	725	18.1%
cell (4,3)	222	717	35.7%
cell (4,4)	199	709	32.4%
cell (5,1)	161	709	26.2%
cell (5,2)	233	717	37.5%
cell (5,3)	0	21	0.0%
cell (5,4)	0	21	0.0%
cell (6,1)	0	21	0.0%
cell (6,2)	217	709	35.3%
cell (6,3)	232	707	37.9%
cell (6,4)	143	701	25.6%
prog logic	10	305	3.3%

Table 7.1: Fault Coverages

Chapter 8

Conclusion

8.1 Research Objectives

We have investigated how to efficiently implement selective fault tolerance by migrating functions into hardware that previously could be implemented only in software. Selective fault tolerance in this context means that the same resources that could be used for duplication-with-comparison or NMR can also be used to support multiprocessing.

Currently, fault-tolerant and multiprocessing systems are not very efficient when the implementation is hardware intensive, or are effected by degraded performance with a software intensive approach. Our objective is to obtain a solution to the problem of fault tolerance and multiprocessing that offers the benefits of hardware and software implementations, while mitigating the costs of both.

A central theme of computer organization is that hardware and software are *logically* equivalent. A designer must decide how to allocate functions to hardware or software, where these decisions are based on such factors as cost, speed, and complexity. In current practice, hardware and software solutions are effectively at extreme corners of the tradeoff space defined by these factors. We have investigated how FPGAs can achieve a more optimum balance between hardware and software tradeoffs in the design of a multiprocessing and fault-tolerant system. Previously, only those operations that were most frequently used, or did not have to deal with different input combinations, had the cost justification of performing them in hardware; now a single FPGA can be frequently changed in order to perform a variety of operations directly in hardware. We have devised an architecture that capitalizes on FPGA technology to offer a high-speed solution to multiprocessing and fault tolerance that is flexible, reliable, and cost-effective.

8.2 Contributions

We have proposed a novel architecture we refer to as Dynamic Reconfigurability Assisting Fault Tolerance (DRAFT). The architecture is comprised of computing modules (CMs), dual-ported result memories (RMs), reconfigurable logic and switching unit (RLSU), an RLSU controller, and ROM. The architecture supports two modes of operation: fault-tolerant (FT) for high-reliability applications, and multiprocessing (MP) for parallel computing applications. Our architecture also allows a multiprocessing application to have replicated processors for combined MP *and* FT operation. Our architecture is flexible; processors do not have to be tightly synchronized, and it combines combinations of applications for efficient and effective processor utilization. Global considerations include tolerating faults in the software by means of functionally redundant, but differently designed, software. Although software faults are design faults, architectural support is required if these faults are to be tolerated efficiently in the field. Voting algorithms for multiple results that are each correct but may not all exactly agree, or performing a reasonableness check on a process, are essential for software fault tolerance. However, until now, a hardware implementation for these functions has not been seriously considered. We have capitalized on the current advances in VLSI by basing the design of the RLSU on Field-Programmable Gate Arrays (FPGAs). The FPGA is dynamically reconfigurable: portions of its logic can change without disturbing the rest of the array. We exploited this property by reconfiguring the FPGA while it operates. Thus, a virtual FPGA is created that is much larger than the physical FPGA. The reconfigurable hardware realizes benefits similar to those software receives from virtual memory management. Through the use of ROM, the FPGA-based RLSU provides *virtual IC* support for fault tolerance and multiprocessing without accruing a significant reliability penalty to the system. Not only does the RLSU offload the overhead from the CMs, it accelerates it.

We analyzed the performance of the proposed DRAFT architecture both analytically and by simulation, and showed that our FPGA-based RLSU design enables greater application throughput than a software-intensive solution that is executed on a very-high speed processor. Simulations were performed hierarchically using VHDL, which is a powerful system-level simulator. Unfortunately, this language supports only deterministic simulation and does not have the necessary capabilities to permit stochastic simulation. We developed a technique for creating randomized application streams that drive the VHDL simulation in such a way as to emulate a stochastic simulator.

Our analysis of DRAFT performance is based on the performance achievable by its components. For processors and memories, performance data such as instruction execution rates and access times, as published in manufacturer's data catalogs, are sufficient to parameterize a performance model. However, FPGA performance is highly dependent on the user's design. Our efforts included design and development of tools to model the FPGA architecture and create FPGA implementations of the DRAFT functions. We accomplished the important task of simulating FPGA dynamic reconfigurability so as to realize the device's full versatility. The FPGA architecture that we chose was one of the first dynamically reconfigurable FPGAs to become available; since then new dynamically reconfigurable FPGA architectures have been announced with new parts releases forthcoming. Our modeling approach is readily applicable to these new architectures so that DRAFT can capitalize on the configurable logic technology of the future. In addition, because of its criticality to our approach, we performed gate-level fault simulations on the FPGA to determine its testability. The experiments determined the effectiveness of "mission vectors" (tests generated for the design that is mapped into the FPGA) for detecting faults in the circuit. It was found that this technique, used prevalently in industry, provides extremely poor fault coverage. It was also found that these circuits are inherently highly-testable, and so the test philosophy concerning these important devices must be changed. The efforts directed towards the FPGA were necessary and contributed to the overall goal of providing a high-performance architecture that supports both fault tolerance and multiprocessing. We have published our findings in numerous symposia proceedings and technical journals [72], [73] [74], [75], [76], [77].

We have analyzed the related issues of cost, scalability, extensibility of operable time, and configurability to provide graceful degradation in the event of processor failures. In all of these analyses, our design has been shown to be superior to comparable designs.

8.3 Future Work

Following this research, the following are areas for further investigation:

- Current design of DRAFT assumes that the proximity of the computing modules to one another is similar to the distances achievable between processors in a multiprocessor. Each of the dual-ported result memories are equally accessible by a processor and DRAFT's reconfigurable logic and switching unit. However, under circumstances where the processors are physically dispersed over an area that cannot be spanned by conventional data buses, then other communication schemes should be considered such as those found in distributed computing environments. By constructing a network of DRAFTs, we could also investigate the use of our architecture as nodes in a fault-tolerant, distributed system.

- In the current design of DRAFT, applications in the arrival queue are treated in a first-come-first-served manner. This restriction can be relaxed to permit reordering of the queue entries to make better use of the CMs. However, now the timeliness property of applications would have to be considered so that deadlines are not exceeded.

- In the current approach, reconfiguration effectively removes a faulty CM so that it no longer participates in the voting. A potential area of investigation is the protocol for restoring the fault-tolerance capability of an application, while it is executing, by switching in an available CM. Accordingly, the RLSU would then be configured to recognize the new processor allocation. There are local application performance issues concerning the migration of one of the surviving redundant processes to the spare CM, and there are global application stream performance issues as an application's processor allocation increases dynamically.

- We have used the FPGA to accelerate functions between processors that support multiprocessing and fault tolerance. The FPGA could also be used as a general custom computing resource that is shared by the processors. In this way, a computation that might otherwise be an overly time consuming operation in software can be carried out directly in the FPGA's specialized hardware. Processor contention for the FPGA is a critical issue because halting a processor while awaiting availability of the FPGA can effectively reduce overall processing speed. Having a processor refrain from using the custom computing facilities and perform the operations entirely by itself, albeit more slowly, could result in higher application throughput. Striking a balance between using the FPGA as a custom computing resource and avoiding processor contention involves extending our analyses to the number and variety of specific operations posed by the application.

- Another exploration involves technology mapping of FPGA designs that make optimum use of the virtual IC concept. With the FPGA we have already exploited partial reconfiguration, configuration parallelism, and overlaying layouts. For the latter, we could develop a systematic approach to recognize similarities between FPGA designs in order to preserve cell programmings. Moreover, we could investigate creating FPGA designs whose structures are homogeneous, so changing between functions involves only a minimum of cell reprogrammings.

Bibliography

[1] von Neumann, J., "Probabilistic Logics and the Synthesis of Reliable Organisms from Unreliable Components," *Automata Studies*, C.E. Shannon and J. McCarthy (eds), Princeton University Press, Princeton NJ, 1974, pp. 43-98.

[2] Babbage, C., "On the Mathematical Powers of the Calculating Engine," (Unpublished Manuscript) Buxton MS7, Museum of the History of Science, Oxford, December 1837, Printed in *Origins of Digital Computers: Selected Papers*, B. Randell (ed), Springer, 1974, pp. 17-52.

[3] Johnson, B., *Design and Analysis of Fault-Tolerant Digital Systems*, Addison-Wesley, 1989.

[4] Siewiorek, D., *et al.*, "C.vmp: The Architecture and Implementation of a Fault-Tolerant Multiprocessor," *Proceeding of the 7th International Symposium on Fault-Tolerant Computing*, IEEE Computer Society, Los Angeles, CA, 1977, pp. 37-43.

[5] Goldberg, J., "A History of Research in Fault-Tolerant Computing at SRI International," *The Evolution of Fault-Tolerant Computing*, A. Avizienis, H. Kopetz, J.C. Laprie (eds), Springer-Verlang, Wein, Austria, 1987.

[6] Lamport, L., *et al.*, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 3, July 1982.

[7] Tanenbaum, A., *Structured Computer Organization*, Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[8] Nelson, V.P., "Fault-Tolerant Computing: Fundamental Concepts," *IEEE Computer,* Vol. 23, No. 7, July 1990, pp. 19-25.

[9] Siewiorek, D.P., and Swarz, R.S., *The Theory and Practice of Reliable System Design,* Digital Press, 1982.

[10] Siewiorek, D.P., "Fault Tolerance in Commercial Computers," *IEEE Computer,* Vol. 23, No. 7, July 1990, pp. 26-37.

[11] Dugan, J.B., Trivedi, K.S., "Coverage Modeling of Fault-Tolerant Systems," *IEEE Transactions on Computers,* Vol. C-38, No. 6, June 1989, pp. 775-787.

[12] Losq, J., "A Highly Efficient Redundancy Scheme: Self-Purging Redundancy," *IEEE Transactions on Computers*, Vol. C-25, No. 6, June 1976, pp. 569-578.

[13] Su, S. Y., and DuCasse, E., "A Hardware Redundancy Reconfiguration Scheme for Tolerating Multiple Module Failures," *IEEE Transactions on Computers*, Vol. C-29, No. 3, March 1980, pp. 254-257.

[14] Kanekawa, N., Maeijima, H., Kato, H., and Ihara, H., "Dependable Onboard Computer Systems with a New Method — Stepwise Negotiating Voting," *The Nineteenth International Symposium on Fault-Tolerant Computing Digest of Papers*, IEEE Computer Society, Chicago, IL, 1989, pp. 13-19.

[15] Hopkins, A., "FTMP — A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," *Proceedings of the IEEE*, Vol. 66, October 1978, pp. 1221-1239.

[16] Wensley, J., "SIFT: The Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," *Proceedings of the IEEE*, Vol. 66, October 1978, pp. 1240-1255.

[17] Rennels, D., "Fault-Tolerant Computing — Concepts and Examples," *IEEE Transactions on Computers*, Vol. C-33, No. 12, December 1984, pp.1116-1129.

[18] Färber, G., "Taskspecific Implementation of Fault Tolerance In Process Automation Systems," *Self-Diagnosis and Fault Tolerance*, M. Dal Cin, E. Dilger (eds), Werkhefte Nr. 4 Attempto Tübingen, 1981.

[19] Ammann, E., Brause, R., Dal Cin, M., Dilger, E., Lutz, J., Risse, T., "ATTEMPTO: A Fault-Tolerant Multiprocessor Working Station: Design and Concepts," *Digest of Papers FTCS-13*, IEEE Computer Society, 1983, p10-13.

[20] Kieckhafer, R., Walter, C., Finn, A., Thambidurai, P., "The MAFT Architecture for Distributed Fault Tolerance," *IEEE Transactions On Computers*, Vol. 37, No. 4, April 1988, pp. 398-405.

[21] Cristian, F., "Understanding Fault-Tolerant Distributed Systems," *Communications of the ACM,* Vol. 34, No. 2, February 1991, pp. 57-78.

[22] Lee, P. A., and Anderson, T., *Fault Tolerance: Principles and Practice*, Second Revised Edition, Springer-Verlag, 1990.

[23] Randell, B., "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975, pp. 220-232.

[24] Avizienis, A., "The N-Version Approach to Fault-Tolerant Systems," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, December 1985, pp. 1,491-1,501.

[25] Laprie, J-C., Arlat, J., Béounes, C., and Kanoun, K., "Definition and Analysis of Hardware- and Software-Fault-Tolerant Architectures," *IEEE Computer*, July 1990, pp. 39-51.

[26] Palumbo, D. L., Butler, R. W., "A Performance Evaluation of the Software-Implemented Fault-Tolerance Computer," *AIAA Journal of Guidance, Control, and Dynamics,* Vol. 9, No. 2, March-April 1986, pp. 175-180.

[27] Walter, C. J., Kieckhafer, R. M., Finn, A. M., "MAFT: A Multicomputer Architecture for Fault-Tolerance in Real-Time Control Systems," *Proceedings of the IEEE Real Time Systems Symposium,* December 1985.

[28] Voges, U., ed., "Application of Design Diversity in Computerized Control Systems," *Proceedings IFIP Workshop on Design Diversity in Action*, Springer-Verlag, Vienna, 1986.

[29] Lala, J. H., and Alger L. S., "Hardware and Software Fault Tolerance: A Unified Architectural Approach," *Proceedings of the 18th International Symposium on Fault-Tolerant Computing,* IEEE Computer Society Press, June 1988, pp. 240-245.

[30] Trimberger, S., "A Reprogrammable Gate Array and Applications," *Proceedings of the IEEE,* July 1993, **81**, (7), pp. 1030-1041.

[31] Kwiat, K., Dussault, H., Debany, W., and Gorniak, M., "Benchmarking 32-Bit Processors Though Simulation of Their Instruction Set Architectures," *Government Microcircuit Applications Conference Digest of Papers*, November 1990, pp. 235-239.

[32] Thambidurai, P., and Trivedi, K., "Transient Overloads in Fault-Tolerant Real-Time Systems," *Proceedings of the Real-Time Systems Symposium*, IEEE Computer Society Press, December 1989, pp. 126-133.

[33] Jagadish, N., *et al.*, "An Efficient Scheme for Interprocessor Communication Using Dual-Ported RAMs," *IEEE Micro*, October 1989, pp. 10-19.

[34] Silberschatz, A., Peterson, J., and Galvin, P., *Operating System Concepts, 3rd Edition*, Addison-Wesley, 1991.

[35] Saunders, L., and Trivedi, Y., "Product Evaluation: VHDL Simulators," *ASIC & EDA*, July 1994, pp. 12-37.

[36] Armstrong, J.R., *Chip-Level Modeling with VHDL*, Prentice-Hall, 1989.

[37] Browne, J. C., "Performance Analysis Drives Choices," *Electronic Engineering Times*, 22 January 1996.

[38] MacDougall, M. H., *Simulating Computer Systems: Techniques and Tools*, The MIT Press, 1987.

[39] Allen, A. O., *Probability, Statistics, and Queueing Theory with Computer Science Applications,* Academic Press, 1978.

[40] Klienrock, L., *Queueing Systems Vol I: Theory,* Wiley, 1975.

[41] Trivedi, K. S., *Probability & Statistics with Reliability, Queuing, and Computer Science Applications,* Prentice-Hall, 1982.

[42] *Core Set of Assembly Language Instructions for MIPS-based Microprocessors*, Version 3.2, 30 October 1987, Maintained by Robert Firth, Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA, 15213.

[43] White, J.A., Hartmann, G. L., Pope, R. E., and Hunger, J. W., "A Multi-Microprocessor Flight Control System," Honeywell Systems and Research Center, Minneapolis, MN, Final Report, Contract No. AFWAL-TR-81-3044, May 1981.

[44] Johnson, B. W., and Julich, P. M., "Fault-Tolerant Computer System for the A129 Helicopter," *IEEE Transactions on Aerospace and Electronic Systems,* Vol. AES-21, No. 2, March 1985, pp 220-229.

[45] Serlin, O., "Fault-Tolerant Systems in Commercial Applications," *Computer* Vol. 17, No. 8, August 1984, pp. 19-30.

[46] Lipschutz, S., *Discrete Mathematics,* Shaum's Outline Series, McGraw-Hill, 1976.

[47] Harper, R. E., Lala, J. H., and Deyst, J. J., "Fault Tolerant Parallel Processor Architecture Overview," *Proceedings of the 18th Fault-Tolerant Computing Symposium,* June, 1988, pp. 252-257.

[48] Baker, S., "Programming Silicon," *Electronic Engineering Time*, February 26, 1996.

[49] *Configurable Logic Design and Application Book*, Atmel Inc., San Jose, CA 1993.

[50] *The Programmable Gate Array Data Book*, Xilinx Inc., San Jose, CA 1992.

[51] *Optimized Reconfigurable Cell Array (ORCA) Data Book*, AT&T Microelectronics, Allentown, PA 1994.

[52] Debany, W. H., Gorniak, M. J., Macera, A. R., Kwiat, K. A., Dussault, H. B., Daskiewich, D. E., "Design Verification Using Logic Tests," *Proceedings of the Second International Workshop on Rapid System Prototyping*, 1991, pp. 17-24.

[53] Cooke, L., "The FPGA Engineering Revolution," *ASIC & EDA*, August 1993, pp. 38-53.

[54] *IEEE Standard VHDL Reference Manual*, IEEE Std. 1076-1987, The Institute of Electrical and Electronic Engineers, Inc., 1988.

[55] Navabi, Z., *VHDL: Analysis and Modeling of Digital Systems,* McGraw-Hill, 1993.

[56] Debany, W. H., Lui, W.-L., Kwiat, K. A., Sherman, K. J., Hayes, J. M., and Carletta, J. E., "Intermediate Form for Digital Model Transformation," *Proceedings of the IEEE Reliability and Maintainability Symposium*, 1986, pp. 11-16.

[57] Crawford, J., "EDIF: A Mechanism for the Exchange of Design Information," *IEEE Design & Test*, February 1985, pp. 63-69.

[58] Hosley, L., and Modi, M., "HITS – The Navy's New DATPG System," *AUTOTESTCON Proceedings*, 1983, pp. 29-35.

[59] Ulsamer, E., "Computers – Key to Tomorrow's Air Force," *Air Force Magazine*, July 1973, pp. 46-52.

[60] Myers, G. J., *Software Reliability: Principles and Practices*, Wiley, 1976.

[61] Wilson, R., "Xilinx Fields Reconfigurable-FPGA Line," *Electronic Engineering Times*, 26 June 1995.

[62] Goel, A. L., and Mansour, N., "Software Engineering for Fault-Tolerant Systems," Rome Laboratory Technical Report, RL-TR-91-15, March 1991.

[63] Burns, A., and Wellings, A., *Real-Time Systems and Their Programming Languages*, Addison-Wesley, 1990.

[64] Foulk, P. W., "Data-Folding in SRAM Configurable FPGAs," *Proceedings of the First IEEE Workshop on FPGAs for Custom Computing Machines,* 1993, pp. 163-171.

[65] Siewiorek, D. P., and Swarz, R. S., *Reliable Computer Systems: Design and Evaluation,* Second Edition, Digital Press, 1992.

[66] Maniwa, R. T., "Reconfigurability: Logical Computing," *Integrated System Design*, June 1995, pp. 18-24.

[67] Brown, S. D., Francis, R. J., Rose, J., and Vranesic, Z. G., "Field-Programmable Gate Arrays," Kluwer, 1992.

[68] Debany, W. H., Kwiat, K. A., Dussault, H. B., Gorniak, M. J., Macera, A. R., and Daskiewich, D. E., "Fault Coverage Measurement for Digital Microcircuits," Mil-Std-883 Procedure 5012, Rome Laboratory (RL/ERDA), Griffiss AFB, NY 13441, Dec. 18, 1989 (Notice 11) and July 27, 1990 (Notice 12).

[69] Debany, W. H., Kwiat, K. A., and Al-Arian, S.A., "A Method for Consistent Fault Coverage Reporting," *IEEE Design & Test of Computers*, Vol. 10, No. 3, September 1993, pp. 68-79.

[70] Spencer T. H., and Savir, J., "Layout Influences Testability," *IEEE Transactions on Computers*, Vol. C-34, March 1985, pp. 287-290.

[71] *MIL-HDBK-217*, "Reliability Prediction of Electronic Equipment," Revision F, Notice 2, Feb. 1995.

[72] Kwiat, K. A., and Hariri S., "Domain-Specific Fault-Tolerant Computing," *Government Microcircuit Applications Conference Digest of Papers*, November 1993, pp. 151-153.

[73] Kwiat, K. A., and Hariri S., "An Architecture for Selective Fault-Tolerance and Multiprocessing Using Field-Programmable Hardware," *Proceedings, 1994 IEEE Dual-Use Technologies and Applications Conference*, May 1994, pp. 195-205.

[74] Kwiat, K. A., and Hariri, S., "Efficient Hardware Fault Tolerance Using Field-Programmable Gate Arrays," *Proceedings ISSAT International Conference on Reliability and Quality in Design,* March 1995, pp. 59-64.

[75] Kwiat, K.A., Debany, W.H., and Hariri, S., "Modeling a Versatile FPGA for Prototyping Adaptive Systems," *Proceedings of the Sixth International Workshop on Rapid System Prototyping,* 1995, pp.174-180.

[76] Kwiat, K. A., Debany W. H., and Hariri S., "Effects of Technology Mapping on Fault Detection Coverage in Reprogrammable FPGAs," *IEE Proceedings: Computers and Digital Techniques*, Vol. 142, No. 6, November 1995, pp. 407-410.

[77] Kwiat, K. A., Hariri, S., and Debany, W. H., "Software Fault Tolerance Using Dynamically Reconfigurable FPGAs," *Proceedings of the Sixth Great Lakes Symposium on VLSI,* IEEE Computer Society, Los Angeles, CA, March 1996, pp. 39-42.

☆U.S. GOVERNMENT PRINTING OFFICE: 1997-509-127-47122

MISSION
OF
ROME LABORATORY

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.